A Thesis submitted for the degree of Master of Science

Haptic Interaction with 3D oriented point clouds on the GPU

L. J. Yanes

September 2015

School of Computing Science

University of East Anglia

Contents

Acknowledgements ix

Abstract x

1 Introduction 1

1.1 Aims and Objectives 2

1.2 Motivation 4

1.3 Challenges 6

1.3.1 Visualization 6

1.3.2 Haptic Interaction 6

1.4 Opportunities 6

1.5 Thesis Outline 7

2 Background & Related work 8

2.1 Virtual Reality (VR) 9

2.2 Computer Aided Design (CAD) 9

2.3 Point-based Models 10

2.4 CUDA based GPU Programming 16

2.4.1 Programming model 16

2.4.2 Memory model 17

2.5 Graphics Rendering 19
2.5.1 Ray Casting 20
2.5.2 Ray Tracing 22
2.5.3 Splatting 23
2.5.4 Texture Mapping Volume Rendering 25
2.5.5 Surface Reconstruction 27
2.6 Haptic Interaction and Rendering 30
2.6.1 Human haptics 32
2.6.2 Machine haptics 36
2.6.3 Computer haptics 41
2.7 Related Work 46
2.8 Discussion 47

3 Point cloud visualization **50**
3.1 From point clouds to 3D surfaces 52
3.1.1 Delaunay Triangulation 52
3.2 Marching Cubes 54
3.3 Point Splatting 61
3.4 Discussion 73

4 Collision Detection **78**
4.1 Spatial Subdivision 80
4.2 CPU Spatial Subdivision 82
4.3 GPU Spatial Subdivision 85
4.3.1 Brute force 89
4.3.2 Two Step Query 90
4.3.3 Dynamic Parallelism 91
4.4 Results 94

5 Haptic Rendering **98**

5.1 Static model single point rendering 101

5.2 Deformable model single point rendering 102

5.3 Results 106

5.4 Discussion 110

6 Conclusions **113**

6.1 System overview 114

6.2 Graphic & Haptic rendering 115

6.3 Limitations & Future work 116

List of Figures

2.1 CAD Software example [43] 10

2.2 GPU Threads organization. The threads are organised in blocks, where communication between different threads is possible. The blocks are organised in a grid of blocks, the only communication mechanism available between blocks is the global memory of the device since they do not share information. [2] 18

2.3 Detailed picture of the different memory spaces available in a CUDA application, on the left hand side, the DRAM with full device available memory, local, global, constant and texture. On the right hand side, the different multiprocessors depicting their registers and shared memories along with the device wide constant and texture caches. [1] 19

2.4 The camera projects rays into the scene passing through the pixel matrix. When a ray intersects with the geometric objects of the scene, the colour information from the object is placed onto the pixel matrix. 20

2.5 Seismic application of ray tracing. The rays are generated from the impulse source and projected onto the different layers of the ground represented by the curved lines, each ray is reflected and refracted by the different layers and captured by the receivers. 23

2.6 On the left hand side, the 3D surface of the object is represented by the dotted lines, the points that represent that surface are referenced as $\mathbf{P}_i$ and $\mathbf{Q}$ is the point of interest. The right hand side of the image, shows the local parametrization of the surface using bases u_0, u_1 and the representation of the different points within this coordinate system as $\mathbf{u}_i$ and the point of interest as $\mathbf{u}$ [44]. 25

2.7 Grounded haptic device (Left) and an ungrounded haptic device. 31

2.8 Image of the parts of the skin, depicting different layers, nerves and tactile receptors [7]. 33

2.9 Stimulus transport system to the brain. On the top of the image is the most internal nerve system, the stimulus ends in the brain cortex after passing through the thalamus via third-order neurons, spinal cord via second-order neurons after being received by the receptors on the skin and transported by the first-order neurons. 35

2.10 Homunculus, a graphical representation of the mapping from the somatosensory cortex occupation to human body areas. 36

3.1 Image showing the quality changes related to the size of the lattice. 60

3.2 The visualisation of the different stages of the splatting algorithm. 65

3.3 Original (a) and blurred (b) depth buffer images. 69

3.4 Visual comparison between the Marching Cubes (a) and the point based rendering technique (b). 72

3.5 Simple cube model image generated using the point cloud rendering technique described in this chapter. 73

3.6 Stanford bunny model surface generated by the point based technique. . . 74

3.7 Cantonese chess piece obtained through CT Scanning, surface generated by the point based technique. 74

3.8 Stanford happy Buddha model surface generated by the point based technique. 75
3.9 Stanford dragon model surface generated by the point based technique. . 76
3.10 Model obtained using photogrammetry. The surface of the rock is rendered using this works splatting technique. 76
3.11 Stanford armadillo model surface generated by the point based technique. 77

4.1 Simple PLY file of a wireframe cube generated using Blender. 80
4.2 Graphical representation of a hash table, where the numbers refer to the indexes of the points in the 3D model. 83
4.3 Showing several threads inserting values into different buckets without synchronisation between them, leading to missing values. 86
4.4 Naive bucket wise synchronisation locks to prevent different threads from inserting into the same bucket at a time, resulting in correct but inefficient insertion. 87
4.5 A sorting based method to insert the values into the buckets. No synchronisation is needed, the result is correct and efficient. 88
4.6 Two step method control flow diagram. 90
4.7 Control flow of parent grid to thread child grid on the device over time. [3] 91
4.8 Control flow of the Naive implementation using dynamic parallelism. . . . 92
4.9 Control flow of the array based query. 94
4.10 Spatial subdivision structure construction time on the CPU and GPU measured in milliseconds, using all the models shown in Table 4.1. 97

5.1 The disk is positioned on top of the surface of the model as a result of the haptic rendering algorithm. The force vector is oriented by the normal of the surface. 100

5.2 The proxy object is oriented by the surface of the cube, the vector represents the distance between the HIP and the proxy. The HIP is represented by a sphere found inside the object below the proxy. 103

5.3 The difference between an unmodified version of the bunny model and a modified one. The image in the bottom depicts the surface of the model after being edited using the haptic tool, an indentation can be observed along the top right of the model and the face. 105

5.4 The visual-haptic system: the user explores the haptic space. He feels a countering force when pushing against the surface of the model. 112

List of Tables

2.1 List of 3D laser scanning devices descriptions, advantages and disadvantages. 14

3.1 Performance of the Marching Cubes implementation using different lattice resolutions 58

3.2 Performance of the point-based rendering implementation using different models run on GeForce 970M. 71

4.1 Hash table construction timings measured in milliseconds using an Intel(R) Core(TM) i7 CPU 4720HQ @ 2.60 GHz and a GeForce GTX 270M. . . . 95

5.1 The hardware used and the timings in ms for the different methods implemented for spatial querying including the force rendering (Brute force, Naive and Array based), the + sign indicates the model is being edited during interaction. 107

5.2 Timings for the methods, using hardware **A**. The times are measured in ms, the + sign indicates the model is being edited during interaction. . . 108

5.3 Timings for the methods, using hardware **B**. The times are measured in ms, the + sign indicates the model is being edited during interaction. . . 108

5.4 Timings for the methods, using hardware **C**. The times are measured in ms, the + sign indicates the model is being edited during interaction. . . 109

Acknowledgements

I would like to thank my supervisor, Dr. Stephen Laycock.

Abstract

Real-time point-based rendering and interaction with virtual objects is gaining popularity and importance as different haptic devices and technologies increasingly provide the basis for realistic interaction. Haptic Interaction is being used for a wide range of applications such as medical training, remote robot operators, tactile displays and video games. Virtual object visualization and interaction using haptic devices is the main focus; this process involves several steps such as: Data Acquisition, Graphic Rendering, Haptic Interaction and Data Modification. This work presents a framework for Haptic Interaction using the GPU as a hardware accelerator, and includes an approach for enabling the modification of data during interaction. The results demonstrate the limits and capabilities of these techniques in the context of volume rendering for haptic applications. Also, the use of dynamic parallelism as a technique to scale the number of threads needed from the accelerator according to the interaction requirements is studied allowing the editing of data sets of up to one million points at interactive haptic frame rates.

1 Introduction

While point-based models gain in popularity in computer applications, given their ease of processing and acquisition, new techniques to generate them, both haptically and graphically are being developed. Today, there are several example applications of point-based models in the field of computer graphics and particularly the gaming industry, where fluids simulations are often based on point data. A point-based object is a model represented by 3D coordinates and possibly associated normals. These point-based datasets can result from laser scanning, RGB-D sensors, mathematical modelling (particle based simulations), among others. Advances in point-based deformable models are being applied for modelling more critical objects such as organ simulations for the health industry to provide medical training to physicians. Representing point-based models as continuous three dimensional objects is also a key challenge in order to broaden its range of applications, with the advances in graphic rendering hardware new techniques are being developed to push the limits of older techniques in creating quality images based on point data. Remote robot operators not only can benefit from the advances in graphic rendering of point data given the nature of RGB-D cameras that generate images which can be directly translated into point clouds, but also from advances in haptic rendering as sensing the remote environment with more than audio-visual cues would definitely improve their understanding of the space they are immersed in. Finally, all these applications can benefit from haptical interaction with editable models or models

that are rapidly changing over time since the world is generally in constant change rather than being static.

It is becoming increasingly difficult to ignore the advances in computer processing power and parallelism, these advances came in the form of new many-core processors and general purpose graphic processing units (GPGPU). For this reason some of the techniques and algorithms developed in the last 10 years are becoming obsolete and are being left behind by the advances in technology. Most of the core techniques in the field of haptic rendering were developed in the past 25 years, and have been scaling well under the premise of a single processor use, but with the introduction to many-core processors these techniques are stalling when it comes to scaling, because current processors have their computing capabilities segmented into multiple cores, that can only be unlocked by the use of parallel programming. In order to achieve the highest possible performance on a current microprocessor the algorithms and techniques for haptic rendering have to be adapted to consider the rapid changes in the technology of current computing architectures for CPUs and GPUs. The motivation of this thesis is to explore the implementation of haptic rendering of large point datasets using the GPU. It is envisaged that the many-core architecture of the GPU will be well suited for constructing the spatial data structures and calculating the haptic feedback for large point datasets in real-time.

1.1 Aims and Objectives

The main aim of this work is the generation of a technique that will take as input 3D point data, and allow the user to visualize the rigid surface and interact with it using a Geomagic Touch haptic feedback device. There are several research questions that need to be answered for interacting with point data using haptics devices as well as fast

modification and visualization of the resulting data. We develop techniques for haptic and graphic rendering to allow real-time interaction with dynamic models based on a meshless approach. The haptic rendering algorithm should be independent of the method used to modify the data. This will enable the haptic rendering approach to be applied to a variety of applications such as virtual sculpting, deformable models or working directly with point data streamed from an RGB-D sensor.

To achieve this main aim the following objectives need to be fulfilled:

1. To implement a haptic rendering algorithm for interacting with point data using a Geomagic touch haptic feedback device.
    a) It must execute within 1ms.
    b) It should support point datasets up to one million points.
    c) It should facilitate modification of the point data during the interaction by not precomputing spatial subdivision data structures.
2. A graphical representation of the point data should render at least at 30 fps.

Visualize the Point Cloud

The data sets will be visualized on the screen with a high level of detail, this visualization would require a reconstruction of the surface of the model, this can be directly obtained from a point-based rendering technique or an explicit mesh reconstruction. The technique will be implemented in the GPU to ensure the interactive capability of the system, also a comparison with the Marching Cubes technique will be presented. The visualisation must run at 30 fps (as stated in objective 2).

Interact with the Point Cloud

Touching the object is a fundamental aspect of this project, which will enable the user to feel the surface of the 3D object through the Geomagic Touch haptic device (as stated in objective 1).

Modify the Point Cloud

The surface of this 3D object will be modifiable during interaction. The techniques for haptic rendering should not precompute any data structures on the point data enabling the points to be updated during interaction.

1.2 Motivation

The motivation for this project is to improve the techniques for haptic rendering and interaction with 3D Point clouds. These improvements potentially have many benefits for haptic related systems or software for example remote sensing of arbitrary environments can be achieved using nowadays popular RGB-D cameras like Microsoft's Xbox Live Vision, also the laser scanning industry is producing more advanced sensors which ultimately result in 3D point clouds.

The gaming industry can benefit from haptic interaction, over time the gaming platforms have been evolving to create more sophisticated virtual worlds and they strive to achieve a more realistic representation of the real world. This can be easily seen in the evolution of computer graphics, in the same sense, haptic interaction has been continuously developed. Most gaming consoles have introduced controller vibration to provide haptic cues to the user about the environment and the status of the player's avatar in-game. Achieving real-time haptic interaction with arbitrary complex environments like those found in

the gaming industry would open new paths for increasing the realism illusion provided by these type of systems. Many of the current techniques for fluid representation are point-based, and hence, developing techniques to represent these models using haptic tools, as well as, on the screen is an interesting area of study from which the gaming industry can benefit.

Supervised training for medical professionals is expensive and inefficient in the way it is handled today. A small number of professionals can be trained at the same time. Moreover, the number of conditions that these professionals can be trained for is limited by the number of cadavers found with these conditions, rendering the assessment and skill transfer processes difficult. Alternatively, virtual environments (VE) can improve selecting, training and certifying physicians. However, developing these systems is a hard task given the requirement to perform force computations within a few milliseconds to achieve realistic simulations. Real-time simulations of 3D point-based organ models have been developed for this purpose [13], it is clear how improving point-based haptic techniques can result in better VEs for medical simulation purposes.

Projects like The Digital Michelangelo Project [34] where 3D point cloud data is obtained from sculptures and then used in side projects like the interactive kiosk in the Galleria dell'Accademia which was installed there since October 28, 2002 can benefit from adding the sense of touch to the visual feedback already provided, thus, enhancing the sensation of immersion and detail.

1.3 Challenges

1.3.1 Visualization

Visualization of 3D point data is an interesting topic in computer graphics, highly dynamic point data resulting from fluid simulations or highly deformable objects is currently being studied and has proved to be a challenging and time consuming task. We study and compare two methods of rendering such surfaces in an interactive environment, point splatting and Marching Cubes.

1.3.2 Haptic Interaction

Haptic interaction with static 3D point clouds has been studied in the past, but models that can be modified are gaining relevance since they provide a better representation of the real world. Some of the studies in haptic interaction use several precomputation and simplification steps in order to achieve haptic rates for models that can be modified during interaction. This study is focused on the design and implementation of direct online rendering methods for 3D point cloud based editable models, this means to work directly with the data from the model, without any precomputations or simplifications of while allowing the model to be edited.

1.4 Opportunities

Current advances in GPU processing power mean that there is an opportunity to not only incorporate touch into human computer interaction (HCI), but also, do so with highly detailed dynamic models. The efficient use of this processing power by haptic rendering and collision detection techniques opens the possibility of general consumer software that

can interact with point-cloud data. Generally commercial haptic feedback devices, come with software based on the principle of single point interaction. The contact between the user and the virtual objects is determined by a single point, generally represented by the tip of a stylus. Considering multipoint interaction is an opportunity to enhance the realism achieved by current methods.

1.5 Thesis Outline

The principal focus of this work is on the graphic and haptic rendering of editable 3D point-based models. In the current chapter the objectives, motivations, challenges and opportunities are defined. Chapter 2 gives an overview of the literature relevant to visual and haptic systems for point-based models, in Section 2.4 a brief introduction to general purpose graphics processing units (GPGPU) programming is provided. Then a compilation of graphic rendering, surface reconstruction and haptic rendering techniques are presented. Chapter 3 focuses on the process of transforming point cloud models into 3D surfaces, presenting the Delaunay triangulation, Marching Cubes (MC) and point splatting techniques. A comparison between the implementations of GPU accelerated Marching Cubes and the point splatting techniques is also examined. Chapter 4 describes a collision detection algorithm for 3D point cloud data comparing CPU and GPU implementations. Furthermore, GPU spatial querying algorithms for the proposed data structure are defined. The haptic force rendering method used and a performance study of the spatial querying is detailed in Chapter 5. Finally, Chapter 6 discusses the impact of the work presented in this thesis and provides further areas of improvement.

2 Background & Related work

To improve the quality of the haptic representation of any virtual object, there are a series of subjects to consider. In this chapter, the main concepts on these subjects are explained in such way that the reader will understand the main issues related to haptic interaction for each of these subjects. The main aspects of virtual object representation can be divided into four different steps: Data Acquisition, Data Representation, Interaction and Data Modification.

Data Acquisition is the process through which a real or imaginary object is represented or translated into a virtual object. This chapter covers this issue in Section 2.1, Section 2.2 and Section 2.3.

The acquisition step generates a virtual representation of an object, in order to translate this representation to one a human can understand is what has been referred to as Data Representation. Data representation in a system incorporating graphic representations presents challenges that are further explained in Section 2.5.

The principal concepts on haptic data representation and interaction are expanded on in Section 2.6.

Finally, data modification in a virtual reality application involving haptic interaction is challenging given the time constraints required to maintain realism. Section 2.4 introduces core concepts on GPU programming which is used in this work to compute

within constraints the interaction and modification of the virtual objects.

2.1 Virtual Reality (VR)

In the context of perception, Virtual Reality simulates a physical environment for the user. This environment recreates an artificial reality, tricking the human senses, for a more convincing immersion in the virtual world.

In practice it is difficult to recreate a convincing VR immersion due to the limitations on data processing speed, given that the more realistic and accurate the models simulating real world objects are, the more data needs to be processed in each cycle of the simulation.

The term haptics is used for any form of communication involving the humans sense of touch. In virtual reality environments haptic technologies have been developed to take advantage of the sense of touch by applying forces, vibrations or motions to the user. This mechanical stimulation can be used to assist in the creation of virtual objects in a computer simulation. A haptic device allows the user to touch, feel, manipulate, create and modify 3D objects simulated in a virtual environment. This human-computer interface can be used to assist in the training of those physical abilities required to use specialized instruments and help in the creation and modelling of virtual objects.

2.2 Computer Aided Design (CAD)

Computer Aided Design is the use of a wide range of computational tools that assist engineers, architects and other professionals with design in their respective activities. These tools can also be separated into 2D drawing software and 3D modelling software. 2D drawing software is based on vectorial geometric entities such as points, lines, polygons

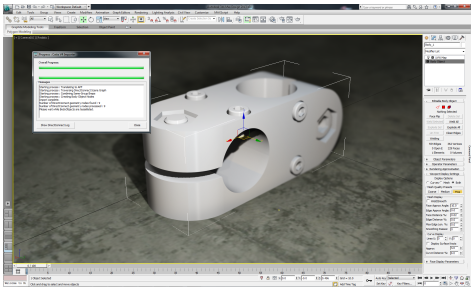


Figure 2.1: CAD Software example [43]

and arches, with which you can interact by the use of a graphical interface. 3D modelling adds surfaces and solids to the list of geometric entities.

In CAD the user can assign to each entity a series of properties such as colour, line style, name and layer, to allow the logical management of the information associated with the model. Also these 3D modelling tools are capable of producing photo realistic rendering of the models, even though it is often the case that these models get exported to programs specialized in animation and visualization of 3D models, for example, Blender, Autodesk Maya or Autodesk 3Ds Max. An example is shown in Figure 2.1

2.3 Point-based Models

3D point-based models can be found in the world of computing as scans of real world objects, for example: to digitize and import an object into CAD software with reverse engineering purposes, to digitally preserve and reproduce an object with historical/scientific purposes, to animate clay models of characters created by artists for entertainment such as for the film or games industry.

In education point-based models are of interest since they provide a simple way to transform objects from the real world into virtual objects that can be analysed or manipulated and placed into specially designed virtual environments with particular

purposes depending on the object. For example a medical oriented virtual environment might be enhanced by including high-detail 3D models of real organs obtained through CT or MRI scans. These scans, after being processed and transformed into 3D virtual models, can be used in educational applications to help students identify and analyse them. More so, the medical field is greatly benefiting from high detail models of specific patient's anatomy to produce custom fitting prosthetics solutions to their problems. There are companies such as Liberating Technologies, Inc. who are leveraging the technological advances in this area, manufacturing state of the art upper limb prosthetic devices.

High resolution 3D point based models can be obtained from 3D scanners. Table 2.1 shows several scanners and their scanning resolutions. As a main subject this work considers haptic interaction with point cloud models, one area of importance is the level of detail needed so as to have our brains believe that the virtual model being touched is a precise representation of the real model. Therefore there is a balance to achieve the correct level of detail to realistically represent a scene. Since most of these scanners are able to build sub-millimetre reconstructions of the scanned objects, and humans have a maximum capacity to distinguish two points in their fingertips at about 2 mm separation [6]. We can safely say that keeping close to 1 mm separation between samples is enough to simulate texture information for the scanned models.

The need for a collision detection and force rendering algorithm that maintains such a level of detail arises in order to haptically interact with these highly detailed models obtained via the 3D scans. These models are obtained from the real world in order to keep specific detail like indentations, bumps, scratches and all kind of imperfections that can be studied in different fields to determine the nature of the object and the different reasons or events that might have occurred to it in order to produce the captured imperfections. Point clouds provide the best choice when trying to describe these objects as other techniques might be prohibitively expensive, a point cloud is a simple structure

that stores the position of a point on the surface of the object and different properties attached to this point (colour, normals, density of points around it, friction coefficient).

Laser scanners generally are devices that work by studying the reflection properties of laser beams directed at an object determining the scanned object surface information. These scanners are generally mounted in a translating/rotating unit in order to capture every angle of the object, the resulting information is stored as a point cloud that represents the surface of the objects along with its different scanned properties.

Dense point clouds consist of tens of thousands to millions of points arranged with an approximate inter-point spacing of 1 mm. Although high detail models can be obtained through 3D scanners it is important to mention that these scanners are not perfect and they are affected by noise, so the raw set of points obtained from the scanning has to be filtered in order to reduce this noise and eliminate outliers from the final point cloud.

Here is a list of some of the available scanners in the market along with their descriptions:

Artec EVA

Similar to a video camera which captures in 3D. Captures 16 frames per second. Real-time automatic frame alignment for easy and fast scanning. Texture capture at 1.3 mp resolution with 24 bpp colour depth.

Next Engine

For desktop size items, given its price and quality, it has good value. Operates with lasers.

MetraSCAN

Very accurate scanning and probing solutions, whether in lab or on the shop floor. It is ideal for large surface metrology and large-scale reverse engineering.

Fuel3D

Costs less than a tenth of its competitors. Great performance and can deal with small to medium sized objects, as well as living subjects.

Mantis Vision F5

Handheld 3D Video Camera for scene modelling, its light weight and good quality but its price is one of the highest.

Faro Laser Scanner

Utilizing non-contact laser technology, generates highly detailed three-dimensional replicas of complex environments and geometries in a matter of minutes.

LDI SLP

High-accuracy, fast, flexible, portable 3D laser scanning system.

Device	Pros	Cons	Resolution	Precision
Artec EVA	Texture capture, acquisition time, light weight.	3D capture of small details.	0.5 mm	0.1 mm
Next Engine	Compatibility with SolidWorks.	Acquisition Time, resolution.	0.2 mm	0.1 mm
Metra-SCAN	Dynamic referencing, complete metrology solution.	Price, small Volumes (210mm x 210mm), connectivity while operating required.	0.1mm	0.085mm
Fuel3D	Value: Quality/Price, portable, texture capture	Ergonomics, not very comfortable to use.	0.25 mm	0.3 mm
Mantis Vision F5	High-resolution, model accuracy, capture time, portability.	Price, only operates indoors.	0.5 mm	0.05 mm
Faro Laser Scanner	Long range, large volume scanning, wide field of view, independent operation, colour camera for photorealism, operates outdoors, mobile scanning	Weight, price, portability.	0.07 mm	2 mm
LDI SLP	High-accuracy, Fast, Flexible, Small size.	Passive optical, no moving parts, narrow angle of incidence.	0.01 mm	0.2 mm

Table 2.1: List of 3D laser scanning devices descriptions, advantages and disadvantages.

There is a wide range of devices that exist for 3D scanning and the number of available devices is growing as the need for them and applications grow, one of the main focuses of the older devices is to virtually register artefacts. But as the quality of the scanners grow other applications start to become more viable, for example the Fuel3D's Eykona product family is focused on wound measurement and registering. Table Table 2.1 shows a list of 3D scanners ranging from relatively expensive 70,000 to cheaper devices of about 1,500. These devices also have a wide range of qualities and capacities, some of them are portable like the Fuel3D and others need to be connected to a PC to work as the MetraSCAN does. The most important factor determining the device quality is the resolution and precision of the device, generally the higher quality devices will have a smaller resolution making them capable of registering high levels of detail with great precision, as the quality of the device lowers the resolution gets higher making this devices less able to capture high detail and display poorer precision.

It is also worth mentioning that even with all of these laser based scanners in the market that can display precise, high resolution models, an alternative way of obtaining a useful representation with up to cm resolution is to use photogrammetry [14] which means to obtain the distance and 3D position information of an object from a set of images. This is a cheap and easy way of making virtual representations of objects since all the equipment that is needed for the reconstruction is a digital camera. On Algorithm 1 a simple algorithm to obtain a point cloud from a physical object is detailed, this technique is used to capture an object as an example on this work, the resulting model can be observed in Figure 3.10.

Algorithm 1 Simplified photogrammetry 3D reconstruction algorithm T

```
Set of uncalibrated images
for Each image I do
    Extract features
end for
for Each feature image do
    Match image features
end for
Apply camera calibrations
Compute 3D points
Output: 3D reconstructed model
```

2.4 CUDA based GPU Programming

General purpose GPU programming is a relatively recent concept in computer science which tries to take advantage and study the computing capabilities of graphics processors.

A GPU is a processor designed for the calculations implied in generating 3D interactive graphics. Some of its characteristics such as; low cost in relation to its computing power, great parallelism and floating point optimizations are considered attractive for its use in applications outside computer graphics, especially in the scientific ambit. GPUs have been used for the implementation of fluid dynamics, clustering algorithms and other applications that require high computing power.

2.4.1 Programming model

The GPU is viewed as a computing device capable of executing a portion of the application under certain assumptions, for example the fact that the same portion of the code has to be executed multiple times on different data and can be written independent of the data. The functions executed in the device are called *kernels*, kernels are executed by the GPU through multiple processors and thousands of simultaneous threads.

The threads are organized into two structures, the block of threads and the mesh of blocks, these are the hierarchical structures in which threads are organized for execution.

The block of threads contains a set of threads that can cooperate together to solve one part of the problem using shared memory. These threads sharing one block can be synchronized. A block of threads can be an array of one, two or three dimensions.

The mesh of blocks is the structure that contains the block of threads, it allows a great number of threads to execute in parallel from just one kernel invocation. In this level the cooperation between blocks can only be achieved by the use of the global memory. The mesh of blocks can only be arrays of one or two dimensions.

Threads, blocks of threads and the grid of blocks can be identified during runtime by using the special identifiers provided by the CUDA language, Figure 2.2 depicts the organization of parallel work on the GPU. The image presents the grid of blocks and the subdivision of each block as a group of threads, the organization of this structure is important for the understanding of the parallel execution model provided by the GPU.

2.4.2 Memory model

The memory used by the GPU is different from the one used by the CPU, so in order to compute a kernel the data has to be copied from the CPU to the GPU. It is important to acknowledge that the bottleneck in GPU programming is the memory transfer to/from the GPU for this reason it is important to keep this in mind while designing algorithms for the GPU in order to process that data as much as possible on the GPU without the need to transfer it.

The GPU has different types of memory optimized for different uses, these are: Global Memory, Shared Memory, Local Memory, Constant Memory and Texture Memory. Global

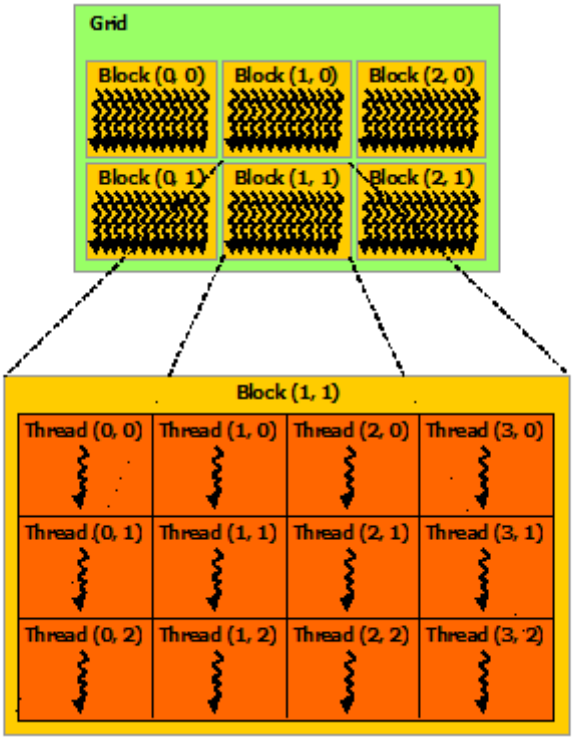


Figure 2.2: GPU Threads organization. The threads are organised in blocks, where communication between different threads is possible. The blocks are organised in a grid of blocks, the only communication mechanism available between blocks is the global memory of the device since they do not share information. [2]

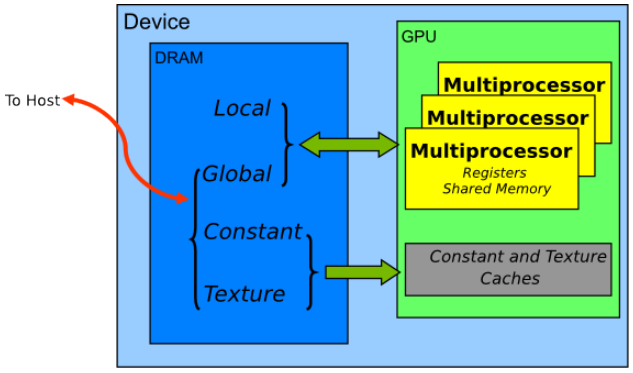


Figure 2.3: Detailed picture of the different memory spaces available in a CUDA application, on the left hand side, the DRAM with full device available memory, local, global, constant and texture. On the right hand side, the different multiprocessors depicting their registers and shared memories along with the device wide constant and texture caches. [1]

memory is for general use on the GPU so it has the greatest space. The access to global memory has to comply to a certain pattern in order to achieve maximum performance of a kernel. The words accessed by the threads on a kernel need to be aligned to 4 bytes and threads need to access this memory in such way that the kth thread accesses the kth word on the memory. The memory model can be viewed in more detail in Figure 2.3.

2.5 Graphics Rendering

The main goal of graphics rendering is to allow the manipulation of the information related to volumes, phenomenon or virtual objects in order to display them. Due to its power to display objects in any way, and observe the internal structure of objects, graphic rendering has proved to be important in the fields of medicine, chemistry, microscopy and astrophysics among others.

Two different styles of rendering exist: Surface Rendering and Volume Rendering. The basic difference between the methods comes from the type of primitives used to represent

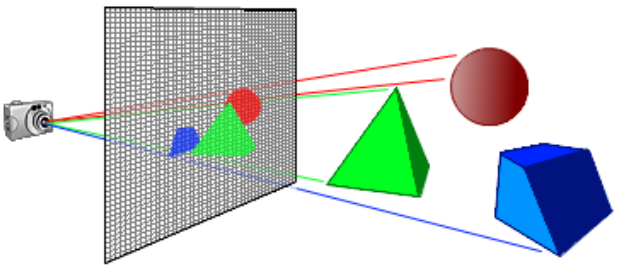


Figure 2.4: The camera projects rays into the scene passing through the pixel matrix. When a ray intersects with the geometric objects of the scene, the colour information from the object is placed onto the pixel matrix.

the volume. For Surface Rendering, the image of the volume is created from the surfaces generated by polygons or contours joined by triangles. In Volume Rendering the volumetric data is directly manipulated without determining an intermediate representation and the information displayed is formed directly from the manipulation of volumetric elements.

There are several different ways of rendering volumetric data, the most common techniques are discussed in this section.

2.5.1 Ray Casting

The first ray casting algorithm was introduced by Appel in 1967 [4]. The main concept of ray casting is to shoot rays from the eye, one per pixel and find the closest object blocking the path of that ray. Then, the colour of each pixel on the view plane depends on the radiance emanating from the visible surfaces as shown in Figure 2.4.

For each pixel in the image $x(t) = o + tv$ represents the equation of the ray passing through it where o is the position of the eye, v is the vector that defines the direction of the ray starting at o and passing through pixel $p(i, j)$.

$$v = \frac{x - o}{||x - o||} \tag{2.1}$$

where x is the floating-point location of the window corresponding to $p(i, j)$ of a discrete view screen of resolution (W, H).

The general algorithm for the ray casting method involves determining the intersection of the casted ray against the surface of the objects in the scene, this can be expressed as:

$$\mathbf{x}(t) = \mathbf{o} + t\mathbf{v} \tag{2.2}$$

Given a surface in implicit form $f(\mathbf{x}) = 0$ where $\mathbf{x}$ is the vector defined by the cartesian coordinates x, y, z:

- Plane $f(\mathbf{x}) = ax + by + cz + d = 0$
- Sphere $f(\mathbf{x}) = x^2 + y^2 + z^2 + d = 0$
- Cylinder $f(\mathbf{x}) = x^2 + y^2 + d = 0$ for $l_1 < z < l_2$

Since all the points on the surface satisfy $f(x, y, z) = 0$ then for any ray $\mathbf{x}(t)$ to intersect the any surface in the scene, the following equation has to be solved.

$$f(\mathbf{x}(t)) = f(\mathbf{o} + t\mathbf{v}) = 0 \tag{2.3}$$

Finally, the normal at the point hit by the ray is computed in order to find the colour of the light returning to the eye along the ray from the hit point and that colour is placed in the corresponding $p(i, j)$ of the image.

2.5.2 Ray Tracing

Ray tracing can be seen as a more general purpose ray casting technique, where the principles are the same as in the ray casting technique described in Section 2.5.1, but has the additional requirement that every ray colliding with a surface produces another single or multiple rays that get traced as well and also contribute to the colour of the pixel $p(i, j)$ represented in the screen.

Ray tracing is generally implemented as a recursive function. The colour of the "initial" ray originated at the camera is computed by collecting the influence of the colour of the first object that this ray touches and the colours provided by all the rays resulting from shadows, reflection or refraction.

The illumination provided by the direct light sources on the geometry being evaluated by the initial ray is then composed with other effects coming from non-direct sources, such as specularity, reflection, refraction or shadows casted by other objects with the aid of the secondary rays. The evaluation of the secondary rays is performed in exactly the same manner as for the initial ray, creating additional sub levels of rays is also possible until certain criteria are met. The different criteria used to stop the recursive generation of rays has a big impact on the performance of the ray tracer, these criteria can be for example a limit on the number of levels of rays or calculating the influence of the reflected or refracted ray that is being generated in order to specify if it is significant enough.

Many algorithms like radiosity use ray tracing for visibility computation, or use the light sources as the starting paths of the rays (photon mapping, light path tracing and others). Ray tracing is also used in applications that are not related to computer graphics, for example in seismic imaging or synthetic seismograms generation [19], [25], where rays are traced from the shot points, and subsequently reflected and refracted at each of the different layers present in the 3D model of the area studied to the geophones for further

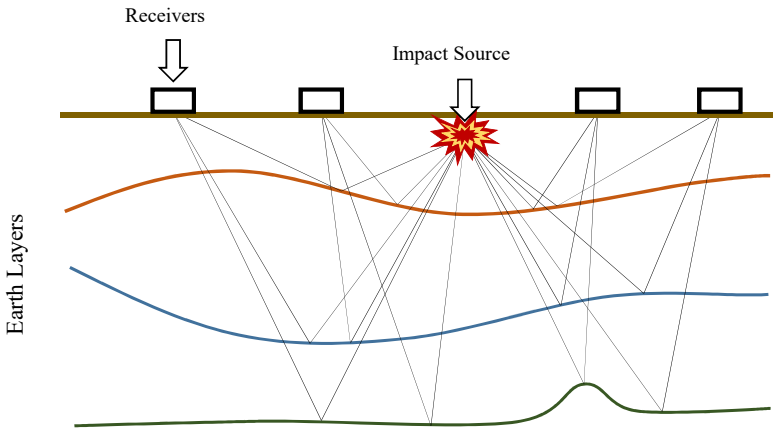


Figure 2.5: Seismic application of ray tracing. The rays are generated from the impulse source and projected onto the different layers of the ground represented by the curved lines, each ray is reflected and refracted by the different layers and captured by the receivers.

analysis of the times required for the rays to travel from the source to the receiver, as depicted in Figure 2.5.

Whited in 1980 [54] was the first to use this form of ray tracing. This recursive form of ray tracing is what is usually associated with the term “ray tracing”.

2.5.3 Splatting

The first idea of using points as a primitive to display 3D objects came from Levoy and Whitted [36]. This technique has been used in contexts such as fluids, fire, smoke and trees rendering. The technique consists of considering each point as a camera facing or oriented 3D disk, where its orientation is determined by the normal in each point. The radius of the disk can be stored along with the normal and colour information or it can be computed during rendering. As a result, each ellipse is mapped to the screen and then

the contribution to the colour of each pixel is determined by the superposition of the different disks that are mapped to the same pixel. The radius is selected so that there is some superposition, thus, generating the perception of a continuous surface.

Point splatting can be accelerated using graphics hardware [9, 11] taking advantage of on-board memory and GPU specific hardware for 3D computations. Botsch et al. [8] proposed GPU accelerated elliptical splatting where a screen-space pre-filter is used, as well as, an object-space reconstruction filter developed by Ren et al [44], providing per-pixel Phong shading for dynamically changing geometries and high-quality antialiasing. Deferred rendering techniques are used in order to simplify the development of custom shaders, by separating the rasterization from the shading of elliptical splats.

Spatial partitioning is a common acceleration technique in computer science, bounding sphere hierarchies, octrees and k-d trees are common examples of these techniques. In 2000 Rusinkiewicz and Levoy [45] used a bounding sphere hierarchy to accelerate the 3D rendering of models containing 100 million to 1 billion samples, achieving interactive graphics rendering rates. Visibility culling, level of detail (LOD) control and rendering are achieved through the hierarchy generation of triangular meshes. The construction is written to disk as a preprocessing step. The use of a kd-tree structure enables interactive rendering rates when the user is moving the camera around the scene by achieving a decimation of points in the original model, but when the camera is static the algorithm recurses into the lower levels of the tree providing more detail to the rendered surface.

Additionally, Sainz and Pajarola [46] developed a view-dependent LOD acceleration technique for point-splatting. Their technique is based on the usage of octrees and also provides a comparison between several point rendering techniques, they state that comparisons have to be based in the observable frame rate (FPS) and not points-per-second rates, given that simple point primitive rendering can achieve 160 million point-rendering rates but when normalized for the observable frame rate, it reduces to 10

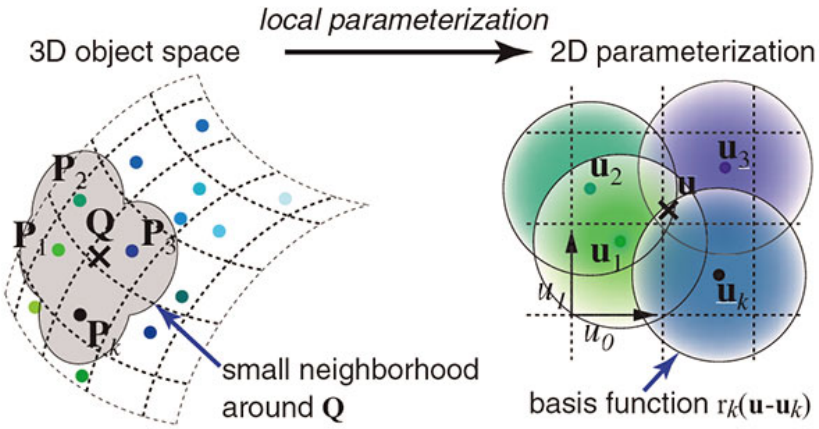


Figure 2.6: On the left hand side, the 3D surface of the object is represented by the dotted lines, the points that represent that surface are referenced as $\mathbf{P}_i$ and $\mathbf{Q}$ is the point of interest. The right hand side of the image, shows the local parametrization of the surface using bases u_0, u_1 and the representation of the different points within this coordinate system as $\mathbf{u}_i$ and the point of interest as $\mathbf{u}$ [44].

million points.

2.5.4 Texture Mapping Volume Rendering

Object aligned slices - 2D Textures

Volume visualization based on texture mapping is a more efficient algorithm than shear-warp for volume rendering, since it takes advantage of the graphics processor's texture hardware. The technique is based on mapping slices of the volume to the visual plane and projecting the results on the screen. In order to speed up the rendering and avoid artefacts, one copy of the volume is generated for each axis, then the most perpendicular to the visualization direction is selected. For the rendering each slice is stacked from back to front composing the values using alpha blending to obtain the final pixel colour, this process is akin to ray casting but having the advantages of GPU hardware.

Since the slices are object aligned, rotating the object produces artefacts on the visualization because of the fact that each "ray" projected from the screen has a different distance from the object than before, this difference is more evident when the angle between the slices and the vision plane is 45 degrees and if the angle gets to 90 degrees then the volume will not be visualized since the visualization rays will be parallel to the slices. This is where the three stack of slices come into play, since the best stack can be chosen depending on the visualization vector. Selecting the most appropriate stack is a matter of deciding which is the maximum component of the visualization vector and then selecting the stack based on this measurement, but the stack change is visible to the user since the interpolation values vary after the stack change, producing a *popping* effect.

The interpolation method generally available in 2D texture rendering is bi-linear interpolation between the samples in each slice, tri-linear interpolation is more complicated on this type of volumetric rendering and requires the use of specialized fragment shaders to be implemented, so that it can take into account more than one slice at a time to get the third coordinate needed for the interpolation.

Vision aligned slices - 3D Textures

Volumetric texture rendering is similar to 2D texture rendering where each value of the volume can be accessed using the (u, v, w) texture coordinates. Volume data is a continuous scalar field where each texture coordinate is in the $[0, 1]$ range. One advantage of volumetric texture rendering is that only one copy of the data is needed for the rendering process. This method uses trilinear interpolation to reconstruct the needed samples. 3D Textures can use hardware tri-linear interpolation improving the volume sampling. If one slice gets assigned arbitrary texture coordinates (u, v, w) in each of its vertices, the hardware interpolates the volume scalars to map them over the slice. This allows the slices to be oriented arbitrarily from the volume.

For 3D texture rendering, the slices are aligned with the vision plane, so if the vision

plane is reoriented, the texture coordinates have to be reoriented for each slice to keep coherence in the visualization. After reorienting the slices must be clipped to only fit the space occupied by the volume. This process is similar to frustum culling where the objects that lay outside of the volume defined by the frustum are not rendered to save time since they are not on the visible part of the scene. Then since the slices can have a different number of vertices after the culling process, the texture coordinates are calculated for each vertex.

Spherical shells have been proposed to avoid the sampling problems of perspective projection against the difference in distance of the volume slices, in order to have constant distances in respect with the eye. The spherical shells are a triangulation of an arch, since this triangulation has to be generated there is a trade-off between a fine or coarse approximation to the curve. This approximation can be appreciated in performance where an inversely proportional relation exists between quality and performance.

2.5.5 Surface Reconstruction

Surface reconstruction algorithms are based on a triangulation of a selected isovalue from the 3D data, generally, medical imaging and other 3D visualization areas, provide an input of a dense field of "density" values, where this density represents the specific field studied. In medical imaging the field values are obtained from CT imaging and then transformed in a way that its easier for the human user to interpret. A useful transformation of the field values is to generate a triangle mesh from them, in order to display that triangle mesh on the screen so that the user can visualize and interpret the data obtained from the CT scan. A known method for the process of producing a triangle mesh from the isosurface of an isovalue of a trivariate function $F(x, y, z)$ is the Marching Cubes (MC) method [37].

To overcome the limitations of the original algorithm, several refinements have been done to the Marching Cubes method and related techniques have been developed.

The asymptotic decider algorithm was developed to solve the ambiguity in some of the cases of the Marching Cubes method, an ambiguous face can be defined according to [40] as a face containing four vertices. Ambiguity then leads to one voxel's face being connected in one manner for a particular voxel and then in a different manner for the other voxel sharing the ambiguous face. The ambiguity is then resolved by extending the original idea in MC of linear variation over the edges to bilinear variation as a mean to resolve the ambiguous faces, by identifying that the ambiguous case is not more than the manifestation of a hyperbola. Selecting whether the data is separated or not by the hyperbola is the criterion in which the conflict resolving of the face is based on.

Marching tetrahedrons (MT) [20], defined a similar framework to the Marching Cubes algorithm and while overcoming the limitations of the patent on the MC algorithm, also resolved the ambiguous cases of MC, and improved the method's viability in a parallel environment. Their method consisted of decomposing the original MC cubes with five tetrahedra, because they do not use six tetrahedra decomposition of the cube, they found a way to make the extracted surface consistent by exploiting the mirror symmetry of the five tetrahedra decomposition one of its planes, so by using an alternating pattern they connect the faces and edges of one group of five tetrahedras with the mirrored version group of tetrahedra in the neighbouring position. To select the tetrahedron for a particular cell, they take advantage of the coordinate location and define a heuristic where the even cells, those of which the x, y, z position's sum is even and odd cells in the case the sum is not even. By making bit operations to the vertex indices based on this heuristic they find all the five tetrahedra's vertices and edges. A bilinear interpolation function on the diagonal of each volumetric grid cube's face is used to locate the position of the vertices in the isosurface. Then a triangulation is made based on the isovalues of

the tetrahedron, and some exceptions are made when the tetrahedron contains one or more 0s as value. It is important to mention that since this method subdivides every cube into 5 tetrahedra the amount of faces that it produces is generally larger than the MC method.

Several Marching Cubes implementations have been accelerated by the use of GPUs, taking advantage of the parallel processing power of the GPU, it is possible to process the MC cube cells in parallel. The challenge becomes in combining the variable number of triangles generated into one continuous mesh. The MT algorithm produces a maximum of two triangles per element, making it advantageous for GPU acceleration since it simplifies the problem of merging the set of produced triangles.

Pascucci [41] uses vertex shaders and the MT algorithm to draw a quad for each tetrahedron using the GPU to interpolate the isosurface over the edges of the tetrahedra in order to place the vertices on the isosurface and determine the gradient of the isosurface to calculate the normal of the faces. After the initial tetrahedra are determined, then a series of successive refinement steps are executed according to the view parameters.

Based on Pascucci's idea and improving over other GPU implementations [27] of the MC algorithm. Dyken and Ziegler [16] developed a high-speed GPU accelerated implementation of Marching Cubes using the histogram pyramids previously developed by Ziegler et al. [57]. The advantages of this technique are the reduced vertex processing that needs to be done, since they provide a hierarchical structure that can cull away the unfilled voxels. Dyken and Ziegler perform the MC algorithm as a combination of expansion/compaction operations. Initially determining the MC algorithm case of each voxel and storing the amount of vertices that said voxel should produce, then the voxels that generate 0 vertices get discarded in a compaction operation and an array of vertices is produced based on the determined amount of vertices for each voxel from the step before. Finally the voxels are populated with the edge-isosurface intersections.

2.6 Haptic Interaction and Rendering

Haptics refers to the sense of touch, haptic technology refers to the process by which a machine provides tactile feedback to a user who receives a force, motion or vibration as a result of his interaction with the machine. There are two types of haptic feedback, force and tactile. Haptic devices are generally different from other usual human-computer interaction HCI devices which are input devices, because haptic devices are input-output devices, the user can provide some input using the device and receive a response resulting from this interaction.

The forces that result from touching an object are what makes our brain understand an object in the physical world if we do not use any other sense. In order to recreate this sensation of touching virtual objects, we have to effect the negative of the force we are applying to an object according to Newton's third law of motion. The haptic device interface provides its position to the computer, so when a user moves the tip of the device in space, this position is updated. The computer has to calculate the adequate force commands for the haptic device, so that an appropriate force is sent to the user, contributing to an approximate haptic perception of the virtual world.

Current developments in haptic technology are leading the path to help people with disabilities interact more accurately with computers, through the enhancement of current graphical user interfaces (GUI). Some technology companies are developing next generation touch screens that allow the users to feel 3D images and manipulate them [12], other mobile device companies (Motorola and Blackberry) use piezo-actuators for the key-click signal delivery.

Tactile feedback refers to the skin sensations, these sensations include temperature, vibration, pressure, pain and surface texture. These are the most common type of device in mobile phones and game controllers, these are typically implemented in the form of

Figure 2.7: Grounded haptic device (Left) and an ungrounded haptic device.

vibration to notify the user of a certain event. Also for hearing impaired people haptic devices can provide useful feedback.

Force feedback refers to the direction and magnitude of the force vectors that can result from interaction with virtual objects such as, rigid bodies, deformable objects and fluids. This type of haptic interaction allows the user to feel the weight, inertia and other physical properties of the objects they are interacting with.

Haptic devices can also be separated as grounded and ungrounded devices, where grounded devices are the ones that have to be attached to a surface like a desk, a wall or the floor. Ungrounded devices are the kind of devices that are not attached to any kind of surface, a good example of an ungrounded haptic device is a gaming platform controller. An image of each device type can be found in Figure 2.7.

According to Srinivasan 1995 [52], haptics studies can be subdivided in three areas:

- Human haptics: Studies human sensing and manipulation through touch.
- Machine haptics: The design of machines to augment human touch.
- Computer haptics: Algorithms and software involved in the rendering of touch of

virtual objects.

2.6.1 Human haptics

This area is focused on the somatosensory system by which we perceive the physical world. Our sense of touch, involves thermal perception, size, roughness, weight, elasticity, viscosity, curvature, angle and orientation.

Feeling the surface of an object involves the mechanoreceptors in our skin epithelia, the main types for humans are Pacinian corpuscles, Meissner's corpuscles, Merkel's discs and Ruffini endings. For a more detailed visualization of these receptors see Figure 2.8.

Each of these receptors has a responsibility, the Pacinian corpuscles are sensitive to high frequency vibrations in the range of 70 Hz to 1000 Hz, aiding in the sense of texture, Pacinian corpuscles adapt rapidly to stimulus. They can be found in the dermis and subcutis (see Figure 2.8). Their receptive area is $101mm^2$. They represent 13% of the hands mechanoreceptors.

The Meissner's corpuscles work most effectively in the 10 Hz to 60 Hz range, giving the best information to our brains about light touch. These cells are highly adaptable which renders them inactive after the initial part of the contact to the stimulus that activated them. They are located in the dermis (see Figure 2.8). Their receptive area is $13mm^2$ on average. The Meissner's corpuscles represent about 43% of the hand's mechanoreceptors. They are not sensitive to temperature.

The Merkel discs respond to stimulus in the frequency range of 0.4 Hz to 100 Hz. This receptors are responsible for feeling surface textures and pressure, responding to skin displacements of 1 μm. Iggo and Muir in 1969 [26] presented their findings on Merkel cells, in the original work they studied these cells from primates and cats. They adapt

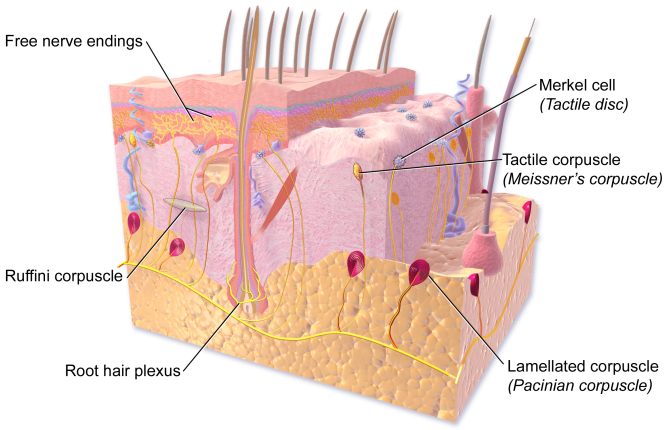


Figure 2.8: Image of the parts of the skin, depicting different layers, nerves and tactile receptors [7].

slowly to stimulus, making them active while the stimulus is present. They are located in the dermis border and epidermis (see Figure 2.8). With an $11mm^2$ receptive area they are the smallest of the mechanoreceptors. They represent 25% of hand's mechanoreceptors.

Finally the Ruffini endings working at 0.4 Hz to 100 Hz frequencies are accounted for the sensation of skin stretch, contributing to finger position and movement control, they also provide angle change sensitivity. They adapt very slowly to stimulus providing continuous sensation. They are located in the dermis (see Figure 2.8). Their receptive area is $59mm^2$. They represent 19% of hand's mechanoreceptors. They have long activation thermoreceptors which provide the sense of feeling heat or burns on the skin.

After all this cutaneous receptors have encoded the sensory information, it gets sent to the central nervous system through the dorsal column-medial lemniscal system and the extralemniscal system. The dorsal column-medial lemniscal system plays the bigger

role out of these two on kinaesthetic and tactile sensory transportation. The information is distributed by two different nerve fibre types, first-order neurons and second-order neurons. The first-order neurons conform the spinothalamic system, which transports pain and temperature, and the lemniscal system is responsible for the mechanoreceptors information transport. The second-order neurons are the link between the spinal cord and the brain (see Figure 2.9).

From the skin receptors, through the lemniscal system, and further to the spinal cord, all the sensory information is transported to an area in the surface of the brain called the somatosensory cortex. The "homunculus" (see Figure 2.10) represents a mapping of the human body to the somatosensory cortex. The larger the area in the somatosensory cortex the higher the sensitivity to stimulus for the corresponding part of the human body.

Studying the course of action that haptic stimuli goes through from a physical point of view is a good beginning to understanding how we process the sensory information perceived by our skin, muscles and organs. How this information is transported to deeper neurons through the nervous system and finally the brain. Also, we can characterize the different type of receptors, stimuli and its courses of action. For example the mechanoreceptors convey all that is pertinent to tactile sensation. As a result, emulating touch stimuli in virtual environments is a field of study that has been promoted by the study of haptic perception.

When we are exploring a surface to identify an object in a real or virtual environment there are several physical attributes of this object that are represented such as contact forces. Haptic perception in virtual environments substitutes the real object with mechatronic machines capable of generating the physical attributes of a real object, these machines are named haptic devices. Haptic devices are responsible for generating a force that accurately represents an object, this process is called haptic feedback.

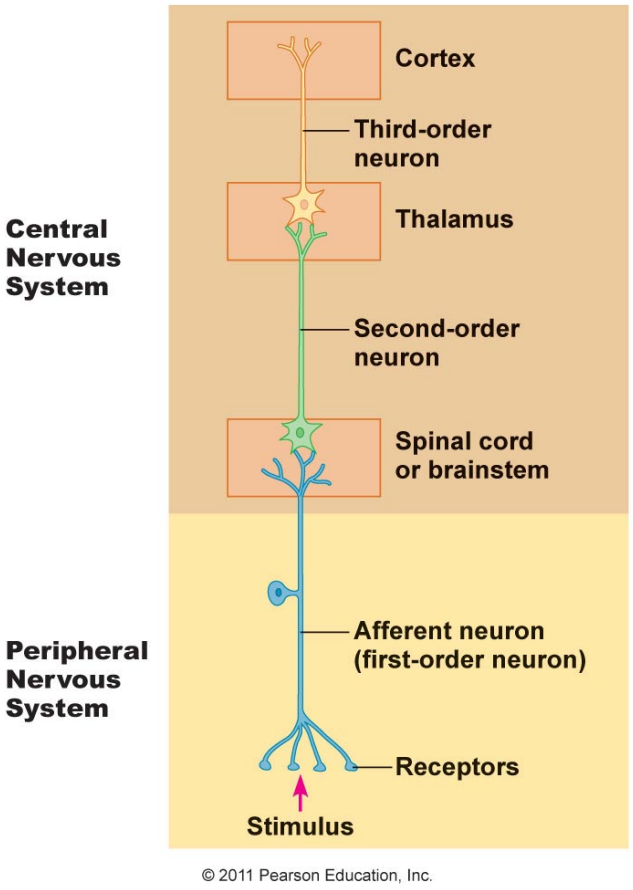


Figure 2.9: Stimulus transport system to the brain. On the top of the image is the most internal nerve system, the stimulus ends in the brain cortex after passing through the thalamus via third-order neurons, spinal cord via second-order neurons after being received by the receptors on the skin and transported by the first-order neurons.

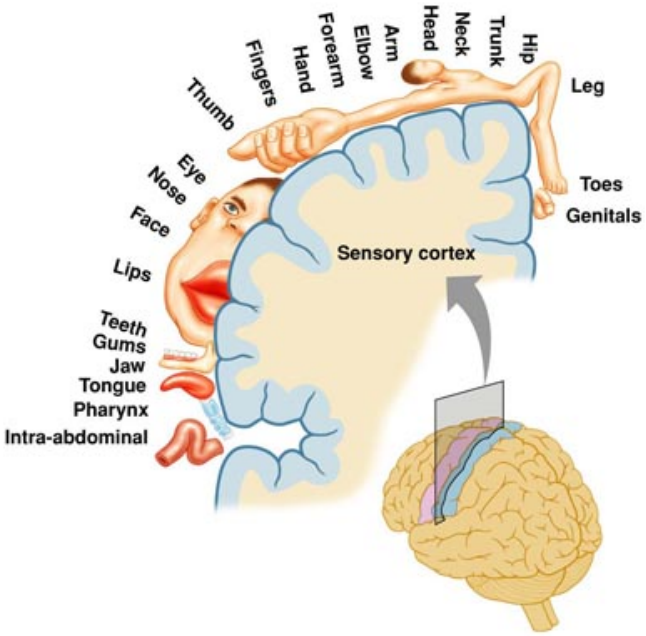


Figure 2.10: Homunculus, a graphical representation of the mapping from the somatosensory cortex occupation to human body areas.

2.6.2 Machine haptics

Machine haptics encompasses all the different aspects of haptic machine design in terms of its operation, shape and size. In addition, machine haptics defines the different attributes of quality for haptic devices.

Haptic interface systems are in their most basic form composed by a power supply, a computer controller and the physical device. The interest of this section lays on the technology that enables haptic devices to provide tactile and kinaesthetic feedback.

A haptic interface is no more than a combination of electromechanical transducers such as: sensors and actuators. Sensors register the interaction between the environment measuring positions and forces and provide the transformation between the real space and the virtual space. Actuators are the mechanisms responsible for providing the motion output in the form of mechanical stimuli over the human body providing as realistic

feedback as possible.

The data in a haptic device flows in both directions between the computer and the real world. The information received from the real world is converted in a analogue-to-digital converter (ADC) to a digital value that can be interpreted by the computer and in a similar manner, the data produced by the computer is transformed by the appropriate converter from digital-to-analogue (DAC). This exchange happens at a very high rate, called Servo Loop Rate, the communication between the computer and the device is highly optimized in order to produce a stable interaction.

Sensors

The sensors in a haptic interface system can be defined as the device that transforms a physical quantity into a signal that can be interpreted by an observer. The output of a sensor is proportional to the physical property that it is measuring, also, sensors are typically made so that they do not interfere with the measured property, but even the most perfect sensor can have a deviation from the true value that it is measuring.

- Sensitivity error: This is a direct measurement of the sensor quality, the lower this type of error the better the sensor, but the prices of sensors also vary in terms of precision, as the device is more precise it becomes more expensive.
- Offset error: when the real property is zero but the sensor outputs a signal that is not zero.
- Dynamic error: if a sensor is digital, not reaching the sampling frequency can cause a dynamic error, or if the noise changes at a periodical frequency close to the sampling frequency.
- Noise: is a random deviation of the signal that varies in time.

- Digitalization error: when the sensor has a digital output, the output is an approximation of the real value of the physical property measured, thus inducing a digitalization error.

Actuators

Actuators are mechanical devices responsible for movement or control in a system. In haptic devices, actuators are the part of the system which moves the parts which exert a force on the user. An actuator in the Phantom Omni is an electric motor. The motor controls the stylus which exerts a force on the user.

- Electrical actuators: These are more commonly motor-based actuators. Among the advantages of these actuators are the ease of installation, relative small size, low levels of noise produced and ease of control. The main disadvantages of this type of actuators are the small force that they can generate, rigidity and low bandwidth.

- Pneumatic actuators: This type of actuators are based on the utilization of compressed air to transfer energy to the haptic interface. They provide higher power ratios than electrical actuators, they are technically simple and lightweight. Disadvantages of this actuators are low bandwidth, stiffness and lubrication problems.

- Hydraulic actuators: They are most commonly based on oil fluids. They provide the highest force output and do not suffer from the lubrication problems found in pneumatic devices. Their disadvantages are high weight and higher levels of maintenance given the fluids need to be filtered and cleaned regularly. Fluid leaks can represent a problem in some environments.

Performance metrics

Haptic device performance and quality is measured in terms of how good of an illusion they can represent a virtual environment. Based on the psychophysical studies presented in Section 2.6.1, these devices have several areas over which their performance can be measured: physical, spatial and temporal. Not only are these metrics important in terms of individual haptic devices but they are also important to take into consideration when planning to acquire a haptic device for a specific application to make sure that it will provide the expected results.

Some of the physical attributes that can be used to define the quality of a haptic device are: inertia, friction, output force, stiffness, backdrivability and size. These attributes were studied in depth by Hayward and Astley [21].

Inertia is measured in grams when its translational and in grams per squared centimetres when its rotational, in order to make the haptic device transparent for the user, the lower the inertia the better is the transparency. Inertia is measured based on the mass of the haptic device, the idea is to avoid introducing extra forces during the process of communication between the real and virtual worlds.

Friction can be measured as static (Coulomb friction) and dynamic (viscous friction or damping). Coulomb friction is independent of velocity and is measured in Newtons (N). Viscous friction is measured as a coefficient in terms of $\frac{\mathrm{N \cdot s}}{\mathrm{m}}$ or $\mathrm{kg \cdot s}$. Both types of frictions are forces that act in the opposite direction of the movement and provide a resistance to free movement.

The maximum output force is the measure in N of the maximum force that the device can display in a short period of time. Continuous force is the amount of force a device can display in an extended period of time. Minimum force is the minimum force that can be displayed by the device allowing for more precise control of the virtual environment.

Force range is the difference between the maximum displayable force and the minimum, the wider the range of forces, the better the device can represent various interaction intuitions.

The stiffness of a device needs to be at least $\frac{25\,\mathrm{N}}{\mathrm{mm}}$ in order for a human user to feel a realistic stiffness behaviour when there is no visual feedback on the task, but is lower when the object is not occluded or invisible.

Backdrivability refers to the ability to have no opposition while moving the haptic device in the workspace. This is measured as the physical friction inside the device as N, the ideal value for this property is zero as it will not provide forces that are not inherent to the virtual environment. In the case of devices with more than one dimension it is possible to have different values of backdrivability for each dimension.

The size of the workspace or the volume in the 3D case in the real world that can be used by the haptic device is a valuable spatial performance measurement. Another important spatial measurement is the position resolution, this is represented by the device as the minimum detectable movement. The number of degrees of freedom (DOF) refers to the number of dimensions over which the device can exert forces, the range goes from non-directional forces (vibration) to 6-DOF. Grounding location is another important spatial measurement since it defines the place where the device is fixed to a static (for example a table or a wall) or semi-static (wrist, shoulder or hips) position. Also, the backslash or dead-space that can be sensed by the user when moving the device in the opposite direction of a force without sensing a change in position is an important spatial measurement.

The temporal attributes that are important in the design of a haptic device are its latency which measures the time taken from the receiving of the command to the response from the device. It does not include the time required by the software to calculate the

command issued to the device. The device bandwidth refers to the frequency that the device can output forces, in Section 2.6.1 the theoretical physical aspects related to noticeable bandwidth were studied but in practice a much higher variation on required device bandwidth has been noticed, ranging from 10 Hz to 10 000 Hz. Peak acceleration is the result of the combination between maximum force and inertia, it depends on the capabilities of the actuators and the inertial properties of the device.

2.6.3 Computer haptics

Computer haptics refers to the software algorithms that translate the virtual environment display and interaction into stimuli in the form of forces exerted on the user. When a person moves their hand in real interaction and meets an object, such an object provides a force response governed by the laws of nature. Computer haptics focusses on processing all the information from the user that is input to the virtual environment and providing feedback that will travel through the person's mechanoreceptors, muscles, and eventually get to the brain where the illusion of interacting with the real world is achieved. Laycock and Day [31] provide a compilation of many techniques employed by computer haptics to enhance the immersive experience exploiting the humans' sense of touch.

Haptic rendering can be categorized depending on the number of dimensions of the virtual object, number of degrees of freedom of the feedback and type of algorithm used for the rendering process. In this subsection we will discuss the software architecture for haptic rendering and some of the algorithms used for haptic rendering, along with their advantages and limitations.

In haptic rendering environments, not only the haptic simulation is of our interest but also the process through which the user receives feedback for each of its inputs to the system. Generally haptic rendering involves a process, through which the information

from the haptic workspace in the real world is translated into virtual coordinates and the forces are processed to provide a realistic immersion. Graphics rendering is used to present the images of the user's actions on the screen, and the physics simulation engine, provides the physical simulation of the virtual environment that the user is interacting with, usually involving collision detection.

The typical haptic software application architecture is defined as three different subsystems that collaborate to create a realistic virtual environment [47]. These systems are:

The Visual System is responsible for presenting the virtual objects on the screen. It contains all the information related to the position of the objects, the camera, and the lights existing in the system. The visual system is created in a separate thread than the haptic system in order to allow for the different update rates. Normally it is implemented as a graphics engine that refreshes the display screen at a rate of 30 Hz.

The Haptic System involves all the logic necessary to obtain the position information from the haptic device, then use this information to calculate the collision with the virtual environment, calculate the forces represented by the input positions from the haptic device and finally implement the control algorithms that will feed the force back to the user in a stable manner. The haptic systems, depending on the type of application, needs to provide the user forces at rates between 300 Hz to 1000 Hz.

Simulation System This system is responsible for updating the positions of the virtual objects, simulating deformable models or doing both. Physically based simulations often require a high processing power to be performed at fast rates, these types of simulations are very computationally expensive and thus, require long processing

times. In order to alleviate the difference between the update rate requirements of haptic applications and the possible update rates of physical simulations, it is common to separate the haptic system and the simulation system in different threads. The simulation system should be implemented in such way that it can update the virtual objects as fast as it is possible, but good results can be obtained with an update rate between 50 Hz to 300 Hz. This systems is partly responsible for collision detection as it updates the positions of the objects in the virtual environment. A simple local representation of the objects closer to the haptic probe can be generated within this system to be fed to the haptic system to perform high rate collision queries.

To perform the tasks described before, computer haptics mainly focuses on the different systems, algorithms and methods that permit us to generate a force presented to the user from a set of constraints represented by a virtual world. In the following subsections some of the popular methods to perform the calculations necessary for correct force shading in a virtual haptic environment are described.

Ray-Based Rendering

Ray based haptic rendering techniques originated from the need of improving upon point based techniques, in order to achieve more realistic haptic rendering results for different types of objects. While maintaining a low cost computational model compared to 3D tools, according to Ho et al (1997) [24] this technique helps users recognise objects faster than simply using a point-based technique. The technique described in [22, 23] by Basdogan et al includes a hierarchical database storing geometrical and material properties of the objects, a collision detection algorithm based on the hierarchical database, a force response model to simulate the interaction between the probe and the objects in the virtual environment and finally a filtering technique for simulating texture and smoothness

of the surfaces. The hierarchical database presented in 1997 by Ho et al. is a binary tree.

Polygon Rendering

Zilles and Salisbury 1995 [58] developed the God-Object technique to render 3D polygon based models with a haptic device. His method improved on previous vector field methods, that suffered from force discontinuity and inability to render small and thin objects. Also improves on methods describing objects by the use of equations, since the method can handle arbitrary geometries. Taking advantage of the humans' poor position sense, and having a historical representation of the haptic interaction point called the *god-object* a correct force is rendered to the user. The god-object is constrained to the surface of the rendered object and the minimum distance between the surface of the object and the current position of the current haptic interaction point is used as a guide for the constrained movement of the god-object.

Voxel Based Rendering

McNeely in 1999 developed a technique based on voxel sampling where the 3D dynamic haptic object is simulated as a point shell [39]. The collision detection is done between the point shell and the voxel mapped static environment. To compute the direction of the forces McNeely found that it was considerably faster to use precomputed surface normals. One of the disadvantages in his method is the discontinuities found in the force magnitude when the points move from one voxel to another. He mitigates the effects of the force magnitude discontinuities by employing a virtual coupling technique. The virtual dynamic object is attached to the real haptic handle by a set of springs to represent translation and rotation and a damper. The virtual object is assigned a mass and then the dynamics of the mass-springs system are solved by integrating the

Newton-Euler equations at a rate of 1000 Hz. The solution of the system constitutes the force that is displayed to the user.

A dexel based method for virtual sculpting was presented by Zhu in 2004 [56]. His systems is based on two principal components, a material to sculpt on and a virtual tool. The material is represented as a set of dexels, which have a smaller memory footprint than the more commonly used voxels in volumetric applications. The virtual tool represents a ball end mill cutter, a flat end mill cutter and a taper end mill cutter. To compute the haptic forces, this techniques takes advantage of the analytical solution to the equations that represent the intersection between a dexel vector (ray) and the tools represented as analytical 3D objects such as cylinder, cone, sphere, parallelogram and pipes. To represent the force the material removal rate (MMR) is considered to be proportional to the dexel removal rate (DRR) and so the output forces are calculated based on this assumption. Given the computational cost of updating the dexels, the haptic cycle and the graphics cycle are separated. The graphics cycle is responsible for representing the dexels on the screen and updating the dexel structure. The haptic cycle is responsible for rendering the forces to the user by interpolating the previously computed force with the current available force, generating a 1 kHz force sample needed for stable haptic interaction.

Sreeni presented a distance field based haptic rendering of scattered oriented points in 2013 [50]. His method is based on pre-computing a distance field on a voxelised version of the scattered oriented points. The distance field is calculated using an indicator function that finds the minimum sphere embedded between a set of points. A 3D grid decomposition for neighbourhood search is employed as an acceleration structure, the resolution of the grid has to be selected according to the rendering requirements. The union of the embedded spheres can be used as an approximation of the object, the indicator function value for the grid nodes inside the union of spheres is 1, while the

grid nodes not contained return a value of 0. This technique can not handle cases where the objects surface is not sampled with the appropriate density, leading to spheres that contain space outside of the object that is being rendered.

2.7 Related Work

The main topics necessary for implementing a realistic haptic simulation are the haptic sensing system and visualisation, A summary of the research with the most influential ideas to the tools that are developed on this work is presented.

The graphics rendering techniques presented in this work are principally influenced by the following publications:

Levoy & Witted [36]: introduced the idea of using the point as a graphic rendering primitive; this opened the doors to visualisation techniques for meshless object representation.

Pfister, et al [42]: improved on the work from Levoy and Witted with an implementation of a rendering technique called Surface Splatting, where the points would be extended to near ellipses and the smoothing techniques were improved to improve on the aliasing problems of the original technique.

Botsch & Kobbelt [9]: published an implementation of GPU accelerated splatting; improved the efficiency of the technique and allowing it to render larger amounts of points at a higher rate, while maintaining the quality of the resulting image.

The most influential publications for this thesis's haptic rendering techniques are the following:

Salisbury, et al [48]: on Haptic rendering of implicit surfaces; presents a framework for

rendering implicit surfaces, the example surfaces used in this work are more suggestive of a constructive solid geometry (CSG) approach. It uses implicit equations to define the shapes with which the user would interact, rather than estimating the implicit surface from data as this work does.

El-Far [17]: on Haptic rendering of point-based models; introduced the concept of haptic rendering of point based models, the collision response technique used in this work is based on the AABB defined by the points rather than using an interpolation over the points. This work starts setting a direction on how to handle point-based data for haptic rendering.

Leeper, Sreeni '12-13 [33, 51]: advances on haptic rendering of surfaces generated using interpolation techniques of point data; Leeper and Sreeni improved on Salisbury and El-Far techniques to introduce techniques that use the information carried by the points along with interpolation techniques and define an implicit surface that can be fitted to the point cloud, resulting in increased precision on the rendered surface.

Kaluschke 14 [29]: GPU accelerated distance querying for collision avoidance; as GPUs gain popularity in many areas that require high-performance, haptic interaction can also be included, although with a high-throughput approach, Kaluschke's work demonstrate how GPUs can be used in a machine haptics environment.

2.8 Discussion

Virtual reality encompasses the human-computer interaction which has a vital role on the users' perception of a simulated environment. There are several difficulties in creating a convincing virtual reality environment, utilizing modern techniques and hardware along with the understanding of the physical system for haptic perception, better virtual

environments can be created.

Point based models are becoming ubiquitous given the technological advances of: RGB-D cameras, laser scanners and point based computer animations. Techniques to represent and interact with point based models are being developed and used in a variety of areas, for example, medical companies have employed points as the primitive for their virtual object models to manufacture prosthetic devices.

The advances in the many-core computing architecture found in GPU processors, has introduced performance advantages for single program multiple data algorithms. Particularly, algorithms that operate on point based models can be implemented in the many-core architecture efficiently, making use of the parallel capacity of the GPU. The development of programming languages such as CUDA has facilitated the task of writing software that targets the GPU effectively, aiding developers in increasing the performance of their applications.

Haptic interaction and visualization go hand in hand when implementing virtual reality simulations, the advantages of using a graphical representation along with haptic interaction cannot be ignored.

Techniques such as Ray casting, Ray tracing and Splatting can be employed to generate a graphical representation of virtual objects defined by points. The marching cubes method along with its variations can also be employed to reconstruct a surface representation of a virtual object.

The main focus of this thesis is on haptic interaction and its importance in generating a virtual environment that exploits a wider range of senses than what is available more commonly. To achieve this, three main aspects of haptic interaction have been studied: Human haptics, Machine haptics and Computer haptics.

Within human haptics, the different receptors on the skin that transmit information to the brain through the nerves and finally to the somatosensory cortex have been studied. The somatosensory cortex within the human brain is where the different parts of the body are represented and the final impulses from the nerve endings on the skins' receptors are processed.

Machine haptics refers to the hardware required to generate the output that the user will feel, also the different components of this hardware. The performance metrics, the decisions behind the design of haptic interaction devices are also studied within machine haptics.

Finally, computer haptics refers to the mathematical models and the different computing techniques required for haptic interaction. The systems requirements along with the limitations imposed by machine performance and the human perception system are highly important in computer haptics, ultimately defining the perceived quality of the haptic interaction.

3 Point cloud visualization

To visualize point clouds, several methods can be used, each of them depend on the requirements of the system. Visualising objects such as those obtained from LiDAR scanners, laser scans of cultural heritage objects or photogrammetry can result in 3D point based data that is often visualised as static data sets and in some cases editing the data set is required, for instance when an artist is creating a 3D virtual model of an animated character.

Several 3D Modelling programs like Blender, Maya, Meshlab and 3D Studio Max require editing the models as the user interacts with the system. 3D modelling programs can use point clouds as input data to generate mesh based surfaces in order to display the data set because, these programs usually have mesh based rendering solutions or share the meshed model with other custom renderers like Pixar's Renderman for display.

This Chapter focuses on the visualization of editable point cloud models along with real-time haptic interaction. Different methods for achieving real-time visualization of point cloud based data are presented.

The main approaches for rendering point clouds can be categorised as follow; if there is a mesh reconstruction or if there is an approximation of the surface based on the spatial information. There are several methods for mesh reconstruction, for example; distance field based methods for implicit surface, ball pivoting and Delaunay triangulation.

Methods where a mesh reconstruction is not needed are called meshless methods, these are based on exploring and estimating the surface properties around the sampled data. In this chapter the different approaches will be explained including, a comparison between two methods, a meshless and a mesh based method. Section 3.2 describes a mesh based approach for 3D point cloud rendering using the Marching Cubes implementation.

Meshless point based rendering methods have gained in popularity in the last ten years, providing an interesting alternative to more traditional mesh based models. They have the intrinsic advantage of providing a fast surface representation of the data since they do not require a mesh to be obtained from the original point cloud which is generally an expensive operation. These methods are particularly popular in fluid rendering, since they can provide a visualization that complies with the visual frame rate required for real-time rendering and also they scale better than mesh based alternatives.

The most common method for point based rendering is based on the point splatting technique. The different implementations of the splatting technique are based on either the architecture that is used; CPU or GPU, or the rendering primitive; Points, Sprites, Triangles, Sequential Points or Sequential sprites [46]. The details of this technique will be explained in Section 3.3.

In this Chapter the process of visualizing a 3D oriented point cloud as a continuous surface will be discussed, and a comparison of the GPU implementation of the Marching Cubes algorithm's performance with regards to rendering quality and speed against an online point splatting algorithm will be given.

3.1 From point clouds to 3D surfaces

Representing a point cloud as a 3D surface is important in order to visualize details of the underlying data of the point cloud effectively. Since points are 0-dimensional geometric objects, we are not capable of understanding what a union of a set of points present in the space are representing to a full extent. Transforming a set of points into a surface can very much improve our understanding of the object that they represent. There are several algorithms that can be used to aid us in this transformation. The first group is based on generating a triangulated surface from the point data. Another approach which does not depend on generating triangle surfaces, and instead, is based on fast re sampling achieved by creating an interpolation kernel that distributes the properties of each point in the set over a defined area.

In this section we will discuss the most common algorithms used to transform a point set into a surface and motivate our choice for comparing two of these algorithms.

3.1.1 Delaunay Triangulation

There are several definitions of what a triangulation is, these definitions come from different points of view. According to the studies of geometry a triangulation of a planar object is the subdivision of this object in triangles. In particular, a triangulation of the surface of an object consists of a net of triangles with points on the given surface covering that surface partially or totally. From a topological point of view, a triangulation can be defined in a more general way, a triangulation of a topological space $\mathbf{X}$ is a simplicial complex K, homomorphic to $\mathbf{X}$ together with an homomorphism $h : K \to \mathbf{X}$.

A triangulation can be evaluated according to these rules:

1. All the triangles of the triangulation have an area bigger than zero.

2. The interior of the intersection between two triangles belonging to the triangulation is always empty.

3. The intersection between two triangles corresponds to an edge or a point.

4. The set of triangles defines a open or closed convex surface.

Terrain modelling, physically based model analysis, differential equation modelling, computer graphics and animation are areas where triangulations are a common method that usually is applied. In triangulation studies, a common concern is to find the "best" triangulation, this can be measured by minimizing the internal angle of the triangles belonging to the triangulation.

A triangulation of a set of points $\mathbf{P}$ can be defined as a Delaunay triangulation if and only if the embedding circle of a triangle in the mesh does not contain any point of $\mathbf{P}$.

Delaunay triangulations are interesting because they provide certain properties that can be demonstrated:

- Minimizes the angle of the triangles in the mesh.
- The union of all the simplices in the mesh is the convex hull of the points.
- The triangulation is unique when no circumscribed circumference contains more than three vertices of the mesh.

A simple Delaunay triangulation with $O(n^2)$ complexity can be implemented by simply testing for every triangle belonging to a valid initial triangulation of $\mathbf{P}$ the validity of its edges. This test is performed by selecting two adjacent triangles $t1, t2$ belonging to the initial triangulation and given p_i and p_j as vertices of the shared edge of $t1$ and $t2$. The edge $\widehat{p_i p_j}$ is illegal if the embedding circle of one of the triangles contains a vertex of the other.

Algorithm 2 Building a Delaunay Triangulation from T

Require: T is a valid triangulation

```
1: function DELAUNAYMESH(T)
2:     Continue ← True
3:     while Continue do
4:         Continue ← False
5:         for all t ∈ T do
6:             if pointInclusionTest(t) = False then
7:                 E ← IllegalEdge(t)
8:                 SwapDiagonals(E, T)
9:                 Continue ← True
10:                break
11:            end if
12:        end for
13:    end while
14: end function
```

Following this method a valid Delaunay triangulation for any point set can be obtained, there are other faster methods to obtain Delaunay triangulations, by employing a divide and conquer approach, Lee and Schachter [32] achieved a computational complexity of $O(n\ log\ n)$.

In the context of this work, it is important to note that this algorithm is not ideal for our application given that is not simple to exploit parallelism with it based on the fact that it works as an incremental process. Due to the restricted time available per haptic frame this method was discarded in an early evaluation stage. A fast implementation of this method that generated a surface from a point cloud in approx. 1 ms for models larger than a couple thousand points was found to be prohibitively slow [18, 10, 30].

3.2 Marching Cubes

In our work reconstructing a continuous surface from a point cloud is a priority towards presenting the user with a clear image representing the object described by the 3D point

data. In order to achieve this transformation, from raw point data into a understandable image of the object, one must use a surface reconstruction technique. In this subsection we will study the application of the Marching Cubes (MC) technique for surface reconstruction of point based models.

The MC algorithm is based on the assumption that the surface to be reconstructed is presented as a 3D lattice of points from which an isosurface will be extracted. This is important to have in mind because the data with which we are working is in a different state, that is, an unstructured oriented point cloud. In order to transform this point cloud to a 3D lattice of "densities" as required by the MC technique we weight the density of the points present in the neighbourhood of the points in the lattice.

$$Density^{lp} = \frac{\sum_{p \in P} w_p * K(p - lp, h)}{\sum_{p \in P} w_p} \tag{3.1}$$

Using Equation (3.1) we can effectively calculate the density of each of the points in the lattice given a radius h. We set the weight of all the points to be equal to the unity. The interpolating function that we are using is an approximation of the Gaussian curve, it is important to note that the K function has to comply with, $\int K(t)dt = 1$ this is to ensure that the estimated values integrate to 1.

The algorithm to convert the point cloud into a volumetric density field that can be processed by the Marching Cubes algorithm consists of the following steps:

1. Provide the point cloud.

2. Create the 3D lattice.

3. Specify the search radius h.

4. For all lattice points

5. Find neighbours of each lattice point.

6. Using Equation (3.1) calculate lattice point density.

7. Normalize and store the results in the 3D lattice.

Algorithm 3 Creating a 3D Density Lattice from P

Require: Unstructured oriented point cloud P

```
1: function calculateDensity3D(P)
2:     Create a 3D lattice of densities D
3:     h ← search radius.
4:     for all d ∈ D do
5:         Neigh_d ← points p in P where dist(d, p) < h
6:         for all nd ∈ Neigh_d do
7:             Density_d ← Density_d + K(p − d, h)
8:         end for
9:         Density_d ← Density_d / w_total
10:    end for
11: end function
```

The Marching Cubes method has been implemented in several different ways. Some of them focus on quality while others focus on performance. Since our work focuses on interactivity, we are making a trade-off with quality to achieve the required frame rates that ensure our system is fast and responsive, in visual and haptic terms. This translates in the need to perform a visual update in approx. 30 ms and a haptic force output in approx. 1 ms. The major trade-off in practical terms regarding the Marching Cubes technique we made was the amount of voxels that are processed.

One of the main challenges of the Marching Cubes algorithm was to keep the meshing and rendering process inside the approx. 30 ms time limit, while also providing haptic feedback of the rendered surfaces to the user. Given that the haptic approach is not triangle based, the triangulation process was not the most favourable for this application

given that this information would be underused.

The algorithm was implemented on the GPU, and contains several stages, first a voxel classification function gets executed for each of the voxels, this classification function evaluates the field values at the corner of each of the voxels and stores the number of vertices that each voxel will generate. The thread configuration for this call is one thread per voxel doing all the voxels at the same time. The function writes the results to two different arrays, a "voxel occupied" array and a "numberOfVertices" array, the first array identifies the voxels that contain geometry, and the second array identifies how many vertices will be needed for the geometry.

In a second step of the algorithm a reduce operation is executed over the "voxel occupied" array to count the amount of voxels that need to be created and this value is stored on the host RAM to be accessed by the CPU.

The next step is to discard all the voxels that will not contain any geometry from the "voxel occupied" array. The idea of removing the empty voxels is based on the optimization of the usage of the GPU, since a thread will be executed per voxel to calculate the geometry contained in each voxel. In order to maximize efficiency in the usage of the device, only the voxels containing geometry will be processed by the function that defines the geometry.

After calculating the voxels that contain geometry the "voxel vertices" array is populated with the start address of the vertex data for each voxel, by using a scan operation.

Finally, the geometry is generated on the "generate geometry" function that runs exclusively on the occupied voxels and knows the amount of triangles needed for each of the voxels, along with their correct addresses. This functions looks up the field values stored in the 3D field data generated before, interpolates the values in order to place the vertices in the correct positions. The Marching Cubes tables are read from 1D textures

Lattice Resolution	Memory	FPS
64x64x64	59 MB	326 FPS
128x128x128	468 MB	75 FPS
256x256x256	723 MB	12 FPS

Table 3.1: Performance of the Marching Cubes implementation using different lattice resolutions

stored in device memory.

Algorithm 4 Generating a surface mesh of a 3D Density Lattice L

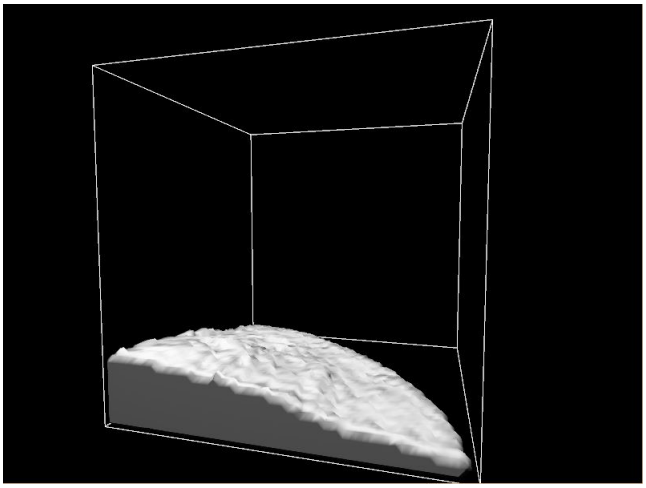
Require: 3D Density Lattice L
Require: GPU stored Marching Cubes edges and triangles table

1: **function** TRIANGULATE3DFIELD(P)
2: Initialize the "voxel occupied" vo array
3: Initialize the "voxel vertices" vv array
4: **for** $i \in$ voxels **do**
5: $vv[i] \leftarrow$ number of vertices required by the MC case given the samples from L
6: $vo[i] \leftarrow (vv[i] > 0)$
7: **end for**
8: $esvo \leftarrow$ **ExclusiveScan(**vo**)**.
9: $numActiveVoxels \leftarrow esvo[numVoxels] + vo[numVoxels]$.
10: **for** $i \in$ voxels **do**
11: $cvo[vo[i]] \leftarrow i$
12: **end for**
13: $svv \leftarrow$ **ExclusiveScan(**vv**)**
14: $numVertices \leftarrow svv[numVoxels] + vv[numVoxels]$
15: **for** $i \in$ voxels **do**
16: Given the cvo and the svv arrays generate the vertices positions and normals interpolating the values of L.
17: **end for**
18: **end function**

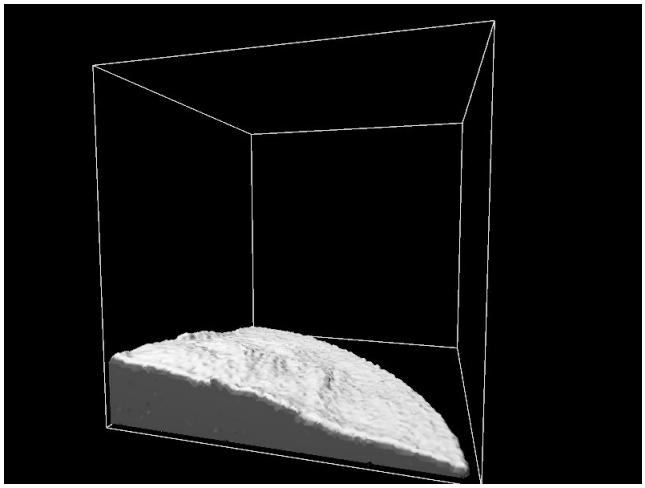
In Table 3.1 we can observe the effect of changing the lattice resolution on the speed and memory footprint required. In the table we observe 3 cases, a 64x64x64 lattice, a 128x128x128 lattice and a 256x256x256 lattice, each of the numbers represents the amount of subdivisions in each dimension, the first number corresponding to X, the second to Y and the third to Z. The memory is measured in megabytes and finally the

FPS represents the frames per second obtained using each different lattice. From the results we can observe that as the lattice resolution is higher the memory requirements increases and the frames per second decreases.

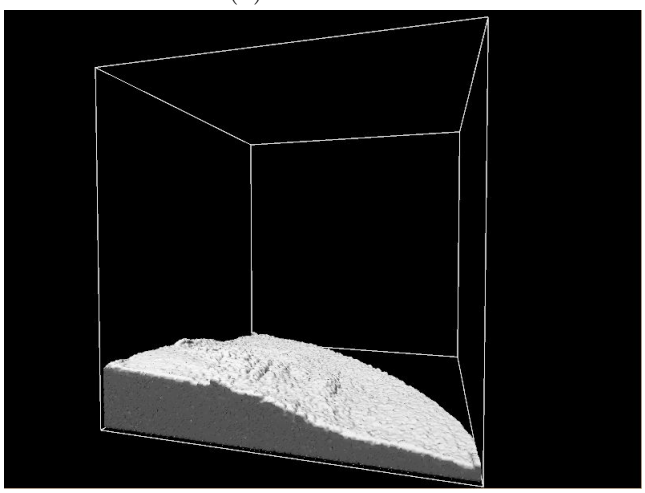
The resulting quality of the meshes generated depend solely on the resolution of the lattice, when the number of points sampled by the lattice corners grows, the smoothness of the resulting mesh grows equally. Computing the normals per vertex of the MC geometry improves the general smoothness of the mesh.



(a) 64^3 Lattice



(b) 128^3 Lattice



(c) 256^3 Lattice

Figure 3.1: Image showing the quality changes related to the size of the lattice.

From these experiments we observed that the Marching Cubes method implemented on the GPU is a good reconstruction technique, but has the downside of increasing memory consumption along with decreased performance when raising the quality of the reconstruction. For this reason we chose not to use this method as our visualization method.

Figure 3.1 shows the differing quality of the mesh and visualization achieved by simply varying the resolution of the lattice. It is easy to observe that as the lattice resolution is larger, the quality of the mesh improves. This has a disadvantage that comes with the MC method, as the resolution of the lattice increases, the maximum number of triangles increases. One consequence of this process is that the triangles produced also get smaller, leading to the possibility of having more than one triangle per pixel, resulting in a slower and inefficient rendering process.

3.3 Point Splatting

This section will provide a detailed description of the rendering process used to display the point based models with the 2D splatting technique. An implementation of the technique is discussed in this section along with results, advantages and disadvantages.

The main idea of the point splatting technique is to render disks in place of the points forming the model in order to obtain a continuous image. These disks commonly referred as splats in the literature, are be orientated according to the point normal if it is known. The image is generated by projecting the points onto the screen and then for each pixel accumulating the contribution from all the overlapping splats, weighted by the distance between the centre of the pixel and the centre of the splat.

The implementation of the point splatting technique is inspired by Botsch & Kobbelt [9],

the system provides an implementation of the renderer in a class called ScreenspaceRendering, which is responsible for displaying a continuous surface of the point based model which is read as a PLY file.

The rendering process begins with the initialization of the ScreenspaceRendering class, this consists of initializing all the buffers and textures that the rendering system needs, these textures are a depth texture, a colour texture, a normal texture, a position texture and another depth texture that will store the blurred depths. Each of these textures will be further explained in the rest of the section.

The depth texture is created so that we can store the depth values during the rendering process, this will allow us to solve two issues, the first is storing the depth values which will be needed in a further step of the rendering algorithm and the second is to provide this value for further rendering systems, in our case, an implementation of a polygonal rendering system is placed over the screen space based rendering system to provide some visual cues for the user, such as a ball and disk model representing the god-object and a ball depicting the haptic interaction point. The depth texture is configured with a nearest filter for both the minification and magnification filters, this means that the depth texture will always provide the value of the closest pixel to the one that is being requested. The wrapping mode of the texture is set up to clamp the values to the edge of the texture. An image is created which is the same size as the initial size of the window where the system is being displayed, for this reason the size of the window is set at the beginning of the program. The format of the texture is a floating point number with the default precision that the graphics driver provides for a `GL_DEPTH_COMPONENT` texture image. Each implementation provides different depths but the standard value for this is a 16-bit floating point number, choosing this value keeps the rendering system compatible with a wide spectrum of graphics cards, from low to high specifications.

The colour texture is created to store the colour attribute of each of the rendered disks,

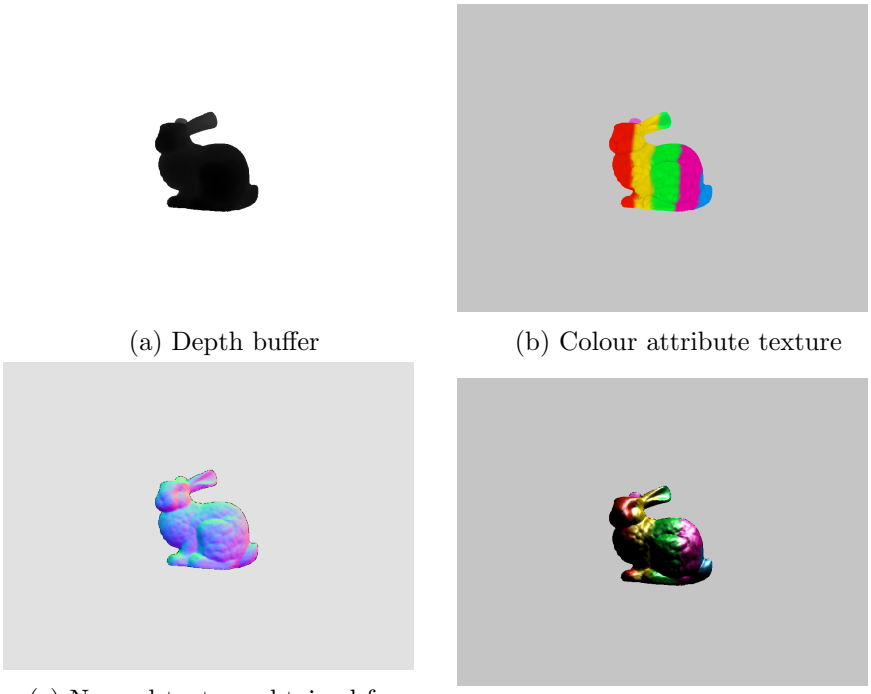
this information is used in combination with lighting in an attribute pass to obtain the final colour. The colour texture has a minification and magnification filter configured to provide the nearest pixel colour value for the position requested. The wrapping is configured to clamp the value to the edge of the texture, so if a value outside the texture is requested, the value obtained will be the same as the value at the edge of the texture. An image with the same resolution of the display window is created to store the colour values of the attribute rendering step. The format of this texture is four single precision floating point numbers, corresponding to the red, green, blue and alpha values of each pixel of the image. This provides the best precision available for this type of texture which is useful in order to blend these values correctly during the rendering algorithm and obtain correct results.

The position texture stores the 2D position of the point being rendered in order to use this information as the base of the calculations for the lighting process where the 3D positions of the pixel, along with its Normal will be important to correctly render the light effects. The position texture is configured as a window sized array of single precision red, blue, green and alpha values, with a minification and magnification filters configured to provide the nearest pixel value of the requested position in the image. The wrapping mode for this texture is set to return the value on the edge of the image whenever a value outside of the image is requested.

The normal texture stores the image space calculated normals for each pixel in the image. The values come from the position texture and the depth texture. In this texture the rendering process is inverted in order to obtain the 3D position of the point blending the position and depths of the different point that were sent through the rendering pipeline, to the apply a screenspace finite differences method to calculate the first derivative of the position field (the surface normal). This texture is configured similarly to the colour texture. The minification and magnification filters are set to the nearest values and

the texture wrap choice is to use the same value as the edge of the texture in case the requested position lies outside of the texture image. The format of this texture is the same as the colour texture with a single precision floating point number representing each of the red, blue, green and alpha channels of the image.

The blurred depth texture is used to store the blurred values of the depth image using a Gaussian filter so that the edges between the different disks represented by the points in the model, rendered on the first pass over the depth texture, are blurred and the result resembles a continuous surface. This texture has a similar configuration as the colour and position textures for the minification and magnification filters the OpenGL flag set is `GL_NEAREST`. The wrap configuration is also the same as the two textures mentioned before, the OpenGL flag set for the wrapping configuration is `GL_CLAMP`. The format of this texture is `RGBA32F` with a single precision floating point value for each of the channels in the image, this is because this image is not used to blur just the values of the depth texture but also the values of the estimated normals calculated from the information in the position and depth textures. The process of softening the normal values resulting from the non softened depth and position textures saves one full blur pass, this is the reason why the values are blurred and the texture has an `RGBA32F` format, where the red channel stores the X component of the normal, the green channel stores the Y component and the red channel stores the Z component, the alpha channel is provided for speedup, since the memory alignment of the graphics cards are generally multiples of 8 bytes, so in this case and for each of the textures even if the alpha channel is not necessary and possibly wasted space, it is faster to create the images with it.



(a) Depth buffer

(b) Colour attribute texture

(c) Normal texture obtained from smoothed depth

(d) Final illuminated image

Figure 3.2: The visualisation of the different stages of the splatting algorithm.

The ScreenspaceRenderer class has a vector of objects `m_objects` that contains all the meshes that need to be rendered, the main program loads the objects that need to be displayed by the ScreenspaceRenderer and inserts them into the `m_objects` vector and when the screen needs to be refreshed it calls the display function of the ScreenspaceRenderer class. This function initially clears the colour and depth buffers, then renders the point's depth, colour and position attributes to the corresponding render targets (the previously described textures). Next the display function blurs the depth texture in order to smooth the overlapping values of the different disks rendered in the previous step. The normal for each pixel in the normal texture is then calculated from the position and blurred depth texture values using the von Neumann neighbourhood of the current pixel. Finally, the colour, depth, normal, and position textures are used in the last rendering pass to output the final image rendered with Blinn-Phong illumination.

Algorithm 5 Screen Space Rendering algorithm

Require: Vector of unstructured oriented point clouds P

1: Clear depth and colour buffers
2: Bind the colour, depth and position textures.
3: Render the points as disks and write the colour, depth and position textures.
4: Bind the Blurred Depth texture.
5: The values rendered to the depth texture are blurred using a Gaussian filter and stored in the blurred depth texture.
6: Calculate the screenspace normals using the blurred depth texture and the position texture rendered in the first pass.
7: Using the normal, depth, colour and position textures render the final lighted image.
8: **Output: Image of the 3D illuminated model represented by P.**

Algorithm 5 depicts the general steps required by the screenspace rendering process in this section we will explain each of these steps in more detail.

The first step of the algorithm described in Chapter 2, involves rendering the points as disks to the screen. Current methods calculate density in a preprocess step to better estimate the size of the disk that needs to be rendered to the screen, with the aim of generating a smooth and continuous surface without empty pixels. However this preprocessing step is not appropriate when the model needs regular updating.

Since the model is assumed to be changing during interaction, to cap the processing power and time requirements, this application compromises quality over speed of rendering by not estimating the density of the points to be rendered to the screen automatically but instead provides a user controlled value for this purpose. The radius of the disk rendered to the screen for each point is defined as a function of this value. To ensure the quality of the final image, the method assumes the input is a dense point cloud. This assumption is important because given this fact an optimal disk radius can be empirically determined to represent the model without holes providing a consistent distribution of point data over the surface of the model.

The disks can be orientated if the normals for each of the points are known, but

also can be facing the viewer if the normals are not known. Again, some applications preprocess the model to determine the normal at each point but given the time and processing power constraints for haptic rendering it was best to not use the processing power nor time to make these calculations since the system is designed to interact with modifying models during interaction. The rendering of oriented disks is ascertained at model loading time if the normals are part of the data in the model.

Listing 1 Vertex Shader for projection of disks

```
vec4 pos = u_ModelView * vec4(Position.xyz, 1.0f);
vec3 posEye = pos.xyz;
float dist = length(posEye);
gl_PointSize = 2 * pointRadius * (1.0f/dist);

PtNormal = u_InvTrans * Normal;
fs_PosEye = posEye;
fs_Position = pos;
fs_Color = vec4(Color.xyz,1.0f);
gl_Position = u_Persp * pos;
```

Listing 2 Fragment Shader for projection of disks and depth computation

```
vec2 ptC = gl_PointCoord- vec2(0.5);
float depth = -PtNormal.x/PtNormal.z*ptC.x - PtNormal.y/PtNormal.z*ptC.y;
float sqrMag = ptC.x*ptC.x + ptC.y*ptC.y + depth*depth;

if(sqrMag > 0.25) { discard; }

vec4 pixelPos = vec4(fs_PosEye + normalize(PtNormal)*pointRadius,1.0f);
vec4 clipSpacePos = u_Persp * pixelPos;
gl_FragDepth = clipSpacePos.z / clipSpacePos.w;

out_Position = pixelPos;
out_Color = fs_Color;
```

The vertex and fragment shaders used for the first pass of the algorithm can be found in Listing 1 and Listing 2.

The second pass of the algorithm is a bilateral filter that is applied to the depth

buffer in order to obtain continuous depth values over the surface represented by the splats. Initially the texture that was rendered in the first pass contains discontinuous depth values, these values need to be smoothed, so a Gaussian filter is employed over the depth texture taking into account the pixels around the pixel to be smoothed within a Chebyshev distance of 2 or 3 units.

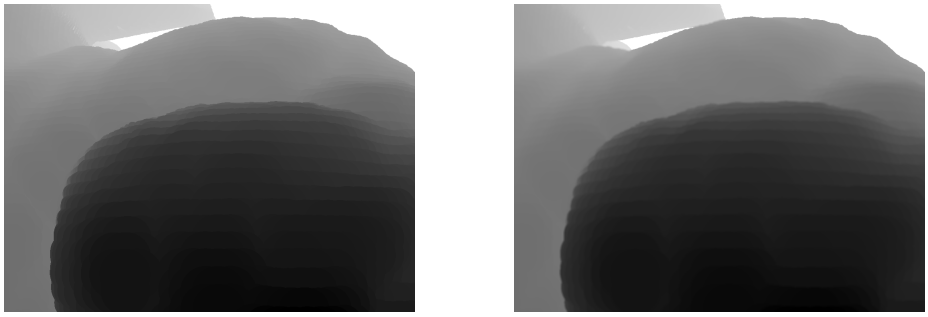
The rendering process of this pass starts by binding the *blurred depth texture* as the render target, attaching the *depth texture* and passing necessary uniform values to the shader. The vertex shader is a simple pass-through to send the pixels of the texture further down the pipeline, leaving the fragment shader to do all the work. Initially the input depth value is obtained from the depth texture, and then this value is linearised using:

$$linearDepth = \frac{2 * near}{(far + near) - depth * (far - near)} \tag{3.2}$$

Where near and far represent the values of the closest and furthest projection planes from the viewer. The fragment shader smooths the depth texture values by sampling the depth texture around each pixel and then applying:

$$blurDepth(x, y) = \frac{\sum depth(x_t, y_t) * e^{x_t^2 + y_t^2}}{\sum e^{x_t^2 + y_t^2}} \tag{3.3}$$

This is a sum over the pixels within 2 or 3 units measured by the Chebyshev distance, where the values x_t and y_t are the different neighbours. This is calculated for every pixel with a depth value smaller than $1 - \epsilon$ for the values closer to 1 the final value is the same as the original depth meaning that these pixels are in the background and not part of the model's surface.



(a) (b)

Figure 3.3: Original (a) and blurred (b) depth buffer images.

A new rendering step is responsible for calculating the model's surface normals, this is a necessary step to correctly calculate the surface illumination. Initially the colour and depth buffer bits of the render target, the *normal texture* in this case, are cleared to avoid having values from the last frame interfering with the result of the current frame. Then the OpenGL program to calculate the normals is activated (the *normal* shader) and the geometry is sent for rendering, the geometry for this step is exactly the same as the geometry used for the previous step. It consists of a screen sized quad divided into two triangles. Next the smoothed depth texture and the position texture are bound to the appropriate shader attributes. Then, the shader parameters are set, these parameters include the near and far plane values, the width and height of the screen, the inverse of the transposed model view matrix and the inverse of the projection matrix. Similarly to the last rendering step, the vertex shader does a simple pass-through over the quad's vertices for further processing of each pixel on the fragment shader.

Finally the normal for each fragment is calculated using:

$$Normal = \frac{F_x(EyePos) \times F_y(EyePos)}{\|F_x(EyePos) \times F_y(EyePos)\|} \tag{3.4}$$

Where F represents the depth values and $EyePos$ is the position of the viewer.

The final rendering step applies the illumination technique, in this case the Blinn-Phong model. To do this, the texture images filled in the previous steps are used, providing all the necessary information to the shader. This step starts with the activation of the final pass shader and setting the geometry that will be rendered. Then the blurred depth texture, the normal texture, the colour texture and the position texture are passed to the shader for these values to be used on the final output. The model view matrix, the projection matrix, the inverse of the model view matrix and the inverse of the projection matrix, the light positions and directions, the near and far planes, and the aspect ratio of the screen parameters are passed to the shader as well.

$$\begin{aligned} I_s &= k_s * I_{incident} * (N \cdot H)^n \\ H &= \frac{L+V}{\|L+V\|} \end{aligned} \tag{3.5}$$

As shown in Equation (3.5) the standard Blinn-Phong model is applied per pixel of the rendered quad, only when the depth stored in the depth buffer texture is bigger than $1-\epsilon$ this equation is not applied to the pixel, since these pixels are outside of the model and belong to the background. The values of L and V are obtained from the light direction and the position texture respectively, k_s is the specular coefficient and $I_{incident}$ is the intensity of the light. This last pass is straight forward as the values that it requires are already stored in the textures that were assigned as parameters and it just needs to calculate the final colour for each of the fragments loading the corresponding values for the different parameters of the equation from either the textures or the shader uniform values.

Point based rendering techniques are gaining popularity as an alternative to triangle based rendering techniques in cases where the data that needs to be rendered comes from depth enriched images, LiDAR or laser based object scans. Given the nature of

Model	Number of Points	FPS
Cube	107409	566
Bunny	208347	227
Archer	598562	315
Happy	543652	110
Dragon	437645	180
Rock	819951	110
Armadillo	1037774	105

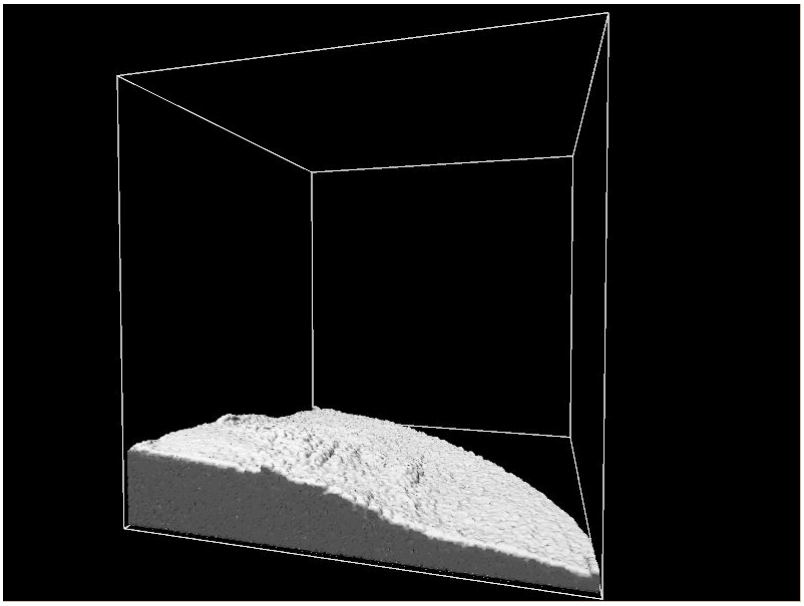
Table 3.2: Performance of the point-based rendering implementation using different models run on GeForce 970M.

the data used in this work (point clouds containing Normal directions), achieving real time graphic refresh rate and adequate rendering quality we hypothesize that point splatting is a viable alternative to surface reconstruction and mesh based rendering. Point splatting has the advantage of minimizing the computational power needed since there is no need to reconstruct the mesh when the data changes and there is a lower amount of operations required each frame for each pixel. There are several advantages that can be leveraged using a point based representation, among others: compact memory footprint, straightforward level of detail mechanisms and easily adjustable rendering quality parameters. Also, this technique can achieve over 100 FPS on a modern GPU for the larger model as shown in Table 3.2.

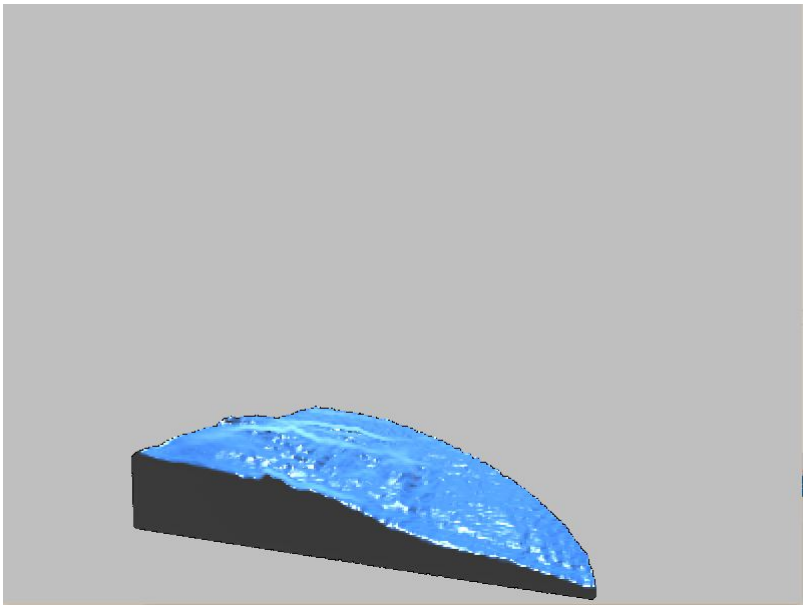
A clear limitation is that, since there is no pre processing step, the best splat size is not dynamically updated nor initially determined leading to poor rendering quality when the density of the points in an area of the screen is too low. Given the time per frame limitations and the number of points that are being rendered, it is very difficult to provide high-quality rendering regardless of the density around each point.

Figure 3.4 depicts an image of the same fluid simulation rendered using the Marching Cubes algorithm, and the point based rendering technique. When using the point based rendering method, the quality of the image depends on the size of the rendered splats,

which can be adjusted by the user according to point density. And, the density of the points on the surface of the fluid.



(a) Marching Cubes.



(b) Point based rendering.

Figure 3.4: Visual comparison between the Marching Cubes (a) and the point based rendering technique (b).

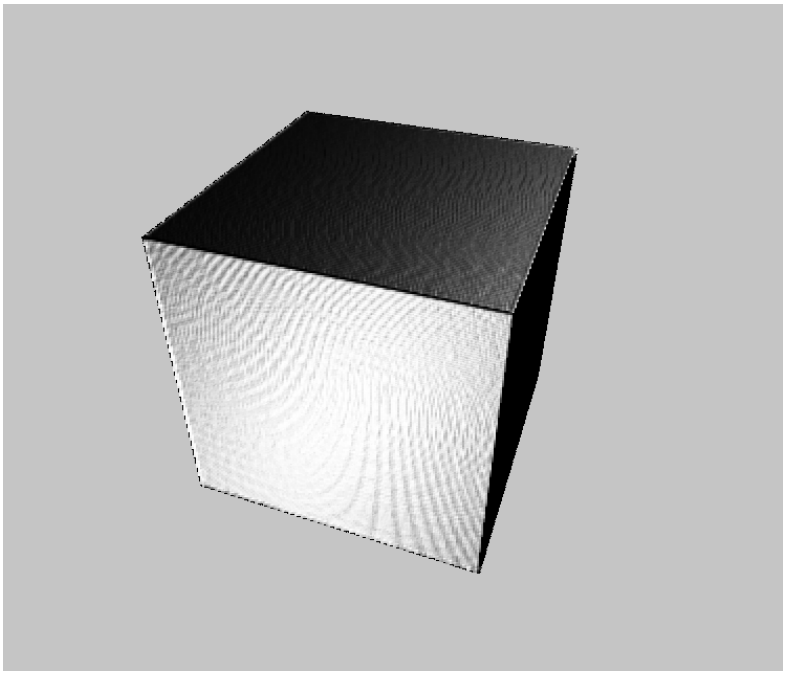


Figure 3.5: Simple cube model image generated using the point cloud rendering technique described in this chapter.

3.4 Discussion

Regardless of the fact that the point based rendering technique presented in this chapter, achieves in excess of 100 FPS for models containing over one million points, the software implementation sets a hard limit to 40 FPS to ensure that there are GPU resources available for the haptic rendering. The limit is necessary given that the implementation is using the same GPU for both the graphic and haptic rendering, an improvement for the system would be to separate the graphics and haptic process to different GPUs in order to increase the resources available for each method. On the other hand, since it is not commonplace to have two GPUs in a computer, the potential number of users would be decreased by this requirement.

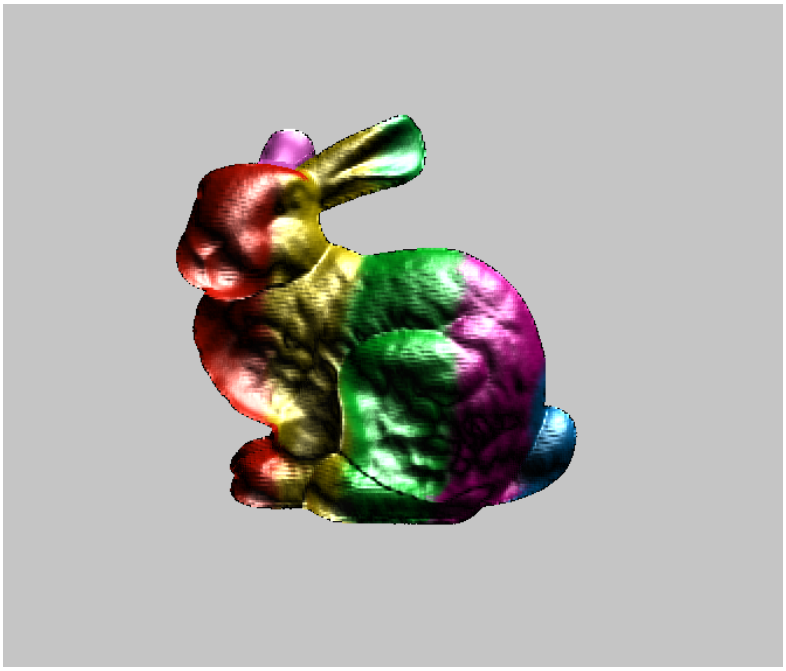


Figure 3.6: Stanford bunny model surface generated by the point based technique.

Figure 3.7: Cantonese chess piece obtained through CT Scanning, surface generated by the point based technique.

Figure 3.8: Stanford happy Buddha model surface generated by the point based technique.

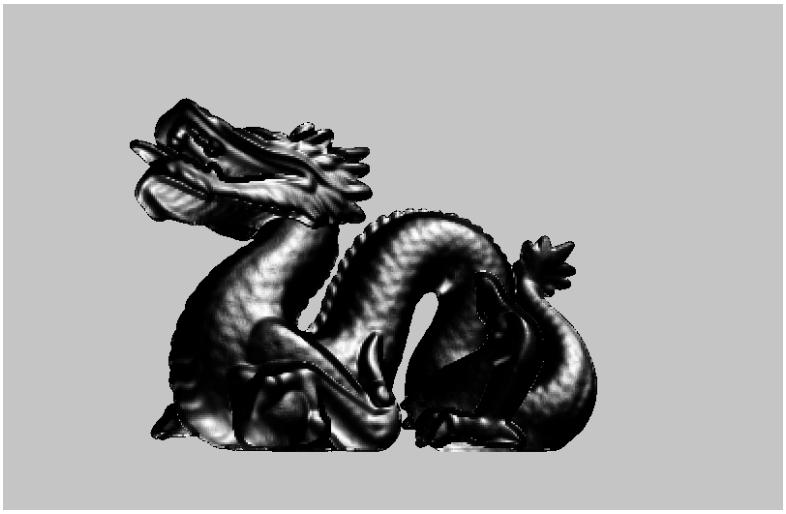


Figure 3.9: Stanford dragon model surface generated by the point based technique.

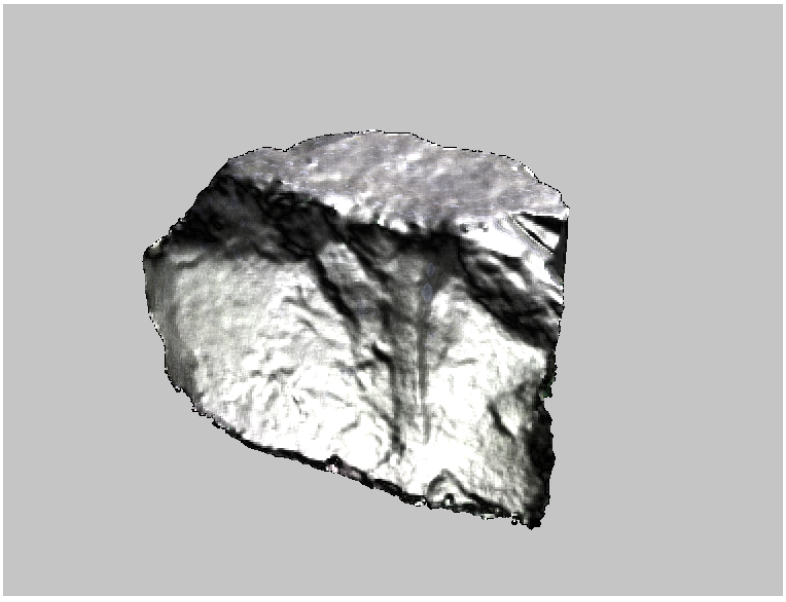


Figure 3.10: Model obtained using photogrammetry. The surface of the rock is rendered using this works splatting technique.

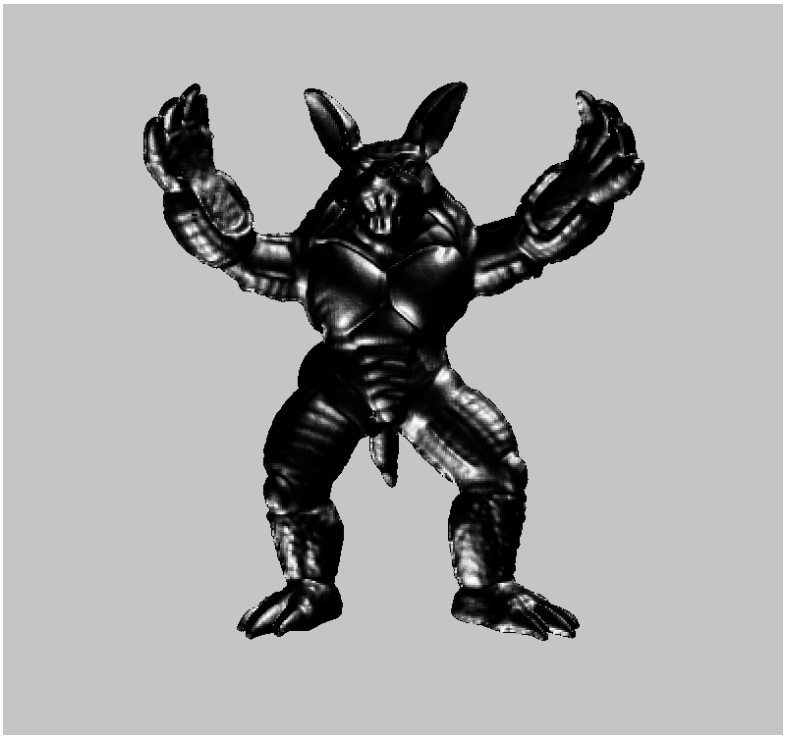


Figure 3.11: Stanford armadillo model surface generated by the point based technique.

4 Collision Detection

This chapter describes our collision detection algorithm for 3D point cloud data. The GPU implementation of the haptic rendering in this thesis is based on the assumption that haptic data is not necessarily static, several approaches have studied the dynamic coupling between multi-rate simulations [15, 53] while the vast majority are based on static simulations. In order to handle direct rendering of point-object interaction within editable models, a fast collision detection framework is needed, which allows spatial subdivision data structures to update at haptic rates. This work incorporates editable objects for haptic interaction, based on the results the potential to incorporate physically realistic models with the algorithms presented in this chapter is possible provided this functionality is implemented on the GPU.

Down-sampling is a common technique used for haptic rendering of dynamic point cloud data [51], thanks to the processing speed provided by the GPUs we can avoid down-sampling and pre-processing the data for scenes of over 100,000 points. In example Table 5.1 we show the haptic rendering timings for a dynamic scene with 208,347 points.

The collision detection framework was implemented entirely on the GPU using the CUDA language, the input data is a VBO loaded as an initialization step and the points are then modified using the direction of the force applied by the user to demonstrate a proof of concept for deformable objects. A similar approach was used by Leeper et al 2012 [33], although their work is based on quasi-static data obtained from a Kinect

device. Given the high volume of points present in the models, we employ a spatial subdivision structure to locate the points that are closer to the HIP. This search for the points close to the HIP is important specially for large models, since it optimizes the search space for the points interacting with the HIP. The spatial structure selected for the neighbour search was spatial hashing based on Teschner et al 2003 [38], we present the timing for the construction of the structure in Section 4.4.

Haptic rendering of deformable models is also a common interest area, in Section 5.2 we study the requirements for the haptic update rate of deformable model interaction, our implementation of the physical simulation achieves haptic rates, so, using a brute force collision detection $O(n^2)$ on the GPU is enough to handle the simulation and computation of the force display.

The spatial partitioning and collision detection algorithm described in Section 4.1 can handle dynamic 3D point data at haptic rates. Haptic interaction rates can also be achieved with editable models, using collision detection implemented on the GPU, a brute force approach is shown in Section 4.3.1.

All of the test models used in the simulation are written in the Polygon File Format (PLY). An example of the file can be seen in Figure 4.1. The larger models are written in binary to reduce the model loading times. This file format is structured as a heading containing in its first line the "ply" keyword, followed by the format of the data in the file. The format can be: ASCII, Little Endian Binary or Big Endian Binary. Then a series of properties for each *element* of the 3D object represented in the file appears next. Each element type is followed by the number of instances of an element. The minimum standard PLY file will always contain a list of *vertex* elements, with X, Y, Z properties, and a list of *face* elements with the vertex_indices property.

The proposed list of standard elements is: vertex, face, edge and material. The

```
ply
format ascii 1.0
comment Created by Blender 2.68 (sub 0) - www.blender.org, source file: ''
element vertex 8
property float x
property float y
property float z
property float nx
property float ny
property float nz
element face 0
property list uchar uint vertex_indices
end_header
-1 -1 -1 0 0 0
1 -1 -1 0 0 0
1 -1 1 0 0 0
-1 -1 1 0 0 0
-1 1 -1 0 0 0
1 1 -1 0 0 0
1 1 1 0 0 0
-1 1 1 0 0 0
~
~
~
~
~
```

Figure 4.1: Simple PLY file of a wireframe cube generated using Blender.

proposed list of properties includes positions, normals, colours and materials for points. For faces vertex_indices, and back face colours are standard. In the case of edges, the standard proposed properties are the index of the vertices constituting the edge and the crease tag if part of a subdivision surface.

Finally, material elements have colour, transparency, reflection, refraction, index of refraction and extinction coefficients. The data of the elements, in the order listed on the header start after the "end_header" line in the numerical format stated on the header.

4.1 Spatial Subdivision

In this section we will study the importance of the spatial subdivision structures for proximity detection. Since our haptic rendering method is based on determining the surface defined by point data nearby the haptic interaction point, a way of identifying these points and their contribution to the surface approximation is required.

Spatial subdivision algorithms are aimed towards decreasing the amount of objects that need to be tested for interaction in a simulation. In order to achieve this reduction some assumptions need to be made, for example, objects that are far from the point of interest do not contribute to the final interaction force. Then we would need to define how much is "far", in our case this is defined by a distance larger than the area of effect of the haptic interaction point.

There are several spatial subdivision structures of which bounding volume hierarchies, binary space partitioning trees, octrees, kd-trees and regular grids are the most popular. All of these algorithms serve similar purposes, but they differ in the way they achieve the purpose. They can be classified in three main categories: Data organization orientated methods, space organization methods and no organization methods [49].

A Bounding Volume Hierarchy (BVH) is a tree structure that contains geometrical objects. The bounding volumes of the objects are grouped by their distances and bounding volumes are built on top of this grouping resulting in a hierarchy of volumes, organized from single object volumes to the largest one that contains all the objects.

A Binary Space Partitioning (BSP) tree is a structure that subdivides the space with a plane separating the data. Each subdivision creates a new plane that can be seen as contained within another separating plane, this gives rise to the hierarchical nature of the structure.

The Octree is a spatial subdivision structure that divides the space of interest into 8 octants, each of these can be further subdivided if a certain heuristic is met, such as if the amount of objects within the octant is greater than allowed or if the space occupied by the objects is larger than a parameters value. When used for point clouds the root node represents the bounding box of the point cloud.

A KD-Tree is similar to the BSP, it is a tree structure that subdivides the space in

half planes, it is useful for range searches, k-nearest neighbours and closest point. A BSP is a generalization of a KD-Tree, the specific property that differs in a KD-Tree from a BSP is that each of the planes that subdivides the space is parallel to one of the axes.

A regular grid is a sparse data structure which subdivides the space in a set number of subspaces for each of the dimensions. It is said to be sparse because all the spaces in the structure are always present whether they contain information or not. This is different from the previously described structures which only keep the spaces that are occupied. There are several ways to implement a regular grid, one of them is creating an array of the same dimensions as the space and setting an amount of buckets for each dimension. Another is to create a one dimensional array of buckets and then generate a mapping function that goes from the dimensionality of the data to the dimensions of the buckets array. A special case of this structure is the hash table where the buckets can be created as they are required providing a more compact set of buckets.

4.2 CPU Spatial Subdivision

In order to deal with large data sets, a regular grid based structure is employed which accelerates the search around the haptic interaction point when identifying the points within a certain region. The efficiency of this structure depends on the number of subdivisions made, this value is equivalent to the amount of buckets that will contain the points within the structure. To optimize the space required by the data structure a hashing technique is employed, where only the buckets containing data are represented. An example of how this structure looks can be seen in Figure 4.2

The hashing structure is implemented in the CPU using the STL [28]. An unordered multimap is responsible for storing the point values in a lookup table. The elements of the table are $< key, Point >$ pairs, where the key is computed using the following

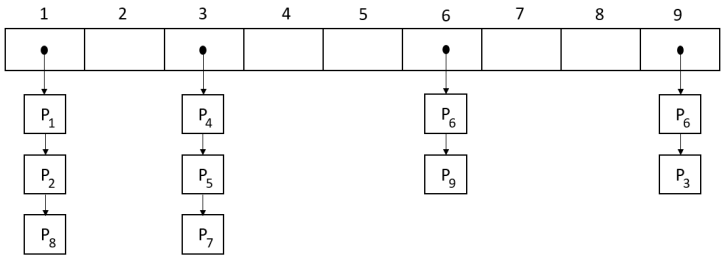


Figure 4.2: Graphical representation of a hash table, where the numbers refer to the indexes of the points in the 3D model.

equations:

$$P' = \frac{P - min}{max - min} \tag{4.1}$$

$$pG = \lceil \frac{P'}{cz} \rceil \tag{4.2}$$

$$key = (pG_x * p1 \oplus pG_y * p2 \oplus pG_z * p3) \mod n \tag{4.3}$$

This set of equations, transforms the original point P from world coordinates into a $[0, 1]$ coordinate system by scaling it using the max and min point in the model. After the point has been scaled its 3D coordinates are located in the appropriate grid position using the cell size, cz. The hash table key is calculated using the hashing function described in [38], where $p1 = 73856093$, $p2 = 19349663$, $p3 = 83492791$ and n is the number of buckets available in the hash table. Once the key is calculated, the $< key, Point >$ pair is inserted in the unordered multimap for later retrieval.

The size of the structure is determined by the loading factor, the parameter for the minimum number of buckets. The size of the hash table is different, depending on the configuration of the table, the number of points which are needed to represent the model

and the density of the points in different places over the surface. The construction time for the hash tables on the CPU can be seen in Table 4.1.

It is important to mention that although for static models, implementing this structure on the CPU can be considered a viable option when we expect the models to change over time, the construction time of this method proves to be insufficient for haptic rendering.

There are several ways that the effects of slow computing times can be mitigated, some authors [5, 39] mention temporal coherence techniques that focus on updating only a part of the model localized to the area of the interaction or modification. One of the main problems of this technique is when the model that is being studied is dynamic and the changes within it do not happen only because of interaction, but because of environmental effect or other parts of the scene, the more moving objects interacting with each other in a scene, the more obvious this limitation is. Temporal coherence is a strong approach when its implementation is single threaded and memory access is not affected by the location of the reads (Random Access Memory), but when subject to multi-threading and the increased costs of random memory accesses on the GPU temporal coherence is less ideal and other approaches need to be considered.

The main advantages of implementing a hash table on the CPU for proximity detection is the ease of implementation, the fast querying of static models regardless of their size, the size of the models that can be represented by this technique (since CPU RAM is generally bigger than GPU RAM) and the portability of the software to almost any conventional processor.

The main disadvantages of the CPU implementation of this method are slow construction times proving to be a limiting factor when the models need to be updated during rendering and the difficulty in the implementation of temporal coherence techniques to address the slow construction times by only modifying the changing part of the structure.

4.3 GPU Spatial Subdivision

The idea behind the need of a spatial subdivision data structure remains the same as described previously, but with the premise that the model will be updated during interaction and so the spatial structure will have to be updated for the proximity detection required during haptic rendering.

It is clear that the CPU version of this process would not suffice in terms of performance. Also, if the model was a computational simulation of a physical object, the computing power of the multiple CPU cores would be largely occupied by the simulation, leaving less computing power available for haptic interaction. For these reasons a GPU implementation of the spatial data structure and proximity detection required for haptic interaction and possibly physical modelling is presented in this section.

To generate a hash based spatial grid on the CPU storing one item at a time is a somewhat simple task. Generating this structure in parallel requires synchronization when inserting the values of interest in the table in order to ensure that there are no race conditions. An example of a race condition might be when trying to insert multiple values in the buckets. If there is no careful synchronization between the threads, some of the values will be lost because the information that each thread reads from the bucket does not consider the other values that are being inserted at the same time, as can be seen in Figure 4.3

The simplest approach to solve this problem would be to have a mutually exclusive lock on each of the buckets (see Figure 4.4) in order to avoid race conditions, but this is not the best solution since the process incurs costly serialization, where for each value that is being inserted in a bucket the lock needs to be checked and if there are other values being inserted the thread will have to wait until the bucket is free. Alternatively atomic operations can be used to have each thread update the bucket without affecting

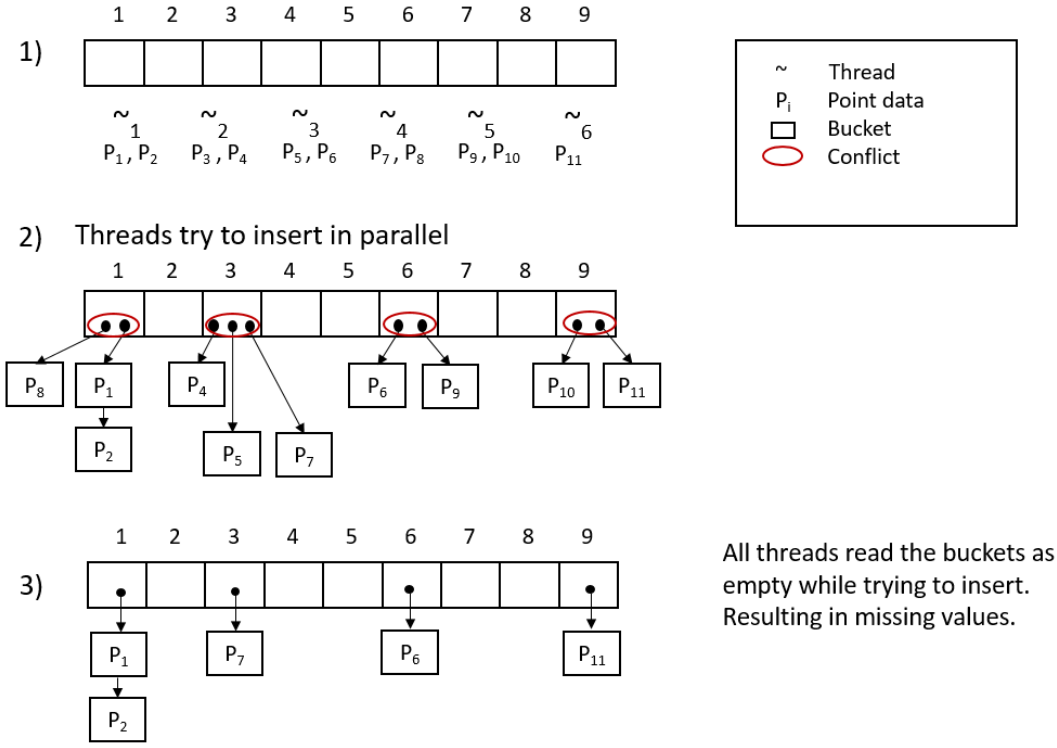


Figure 4.3: Showing several threads inserting values into different buckets without synchronisation between them, leading to missing values.

the other threads.

To avoid race conditions and increased synchronization costs, a sorting method is implemented. The idea behind the algorithm is to sort the points array according to their bucket (from lower bucket id to higher). Then, arrays indicating the start and end of the buckets are generated from the ordered particle lists. Finally the properties of the points are sorted using the sorted id arrays in order to improve lookup locality and cache reuse, see Figure 4.5.

The GPU implementation of the uniform spatial grid, requires extra data beside the point array. The extra required arrays are, a hash value per point, a sorted index per point and the bucket start and end arrays. The total memory requirements depends on

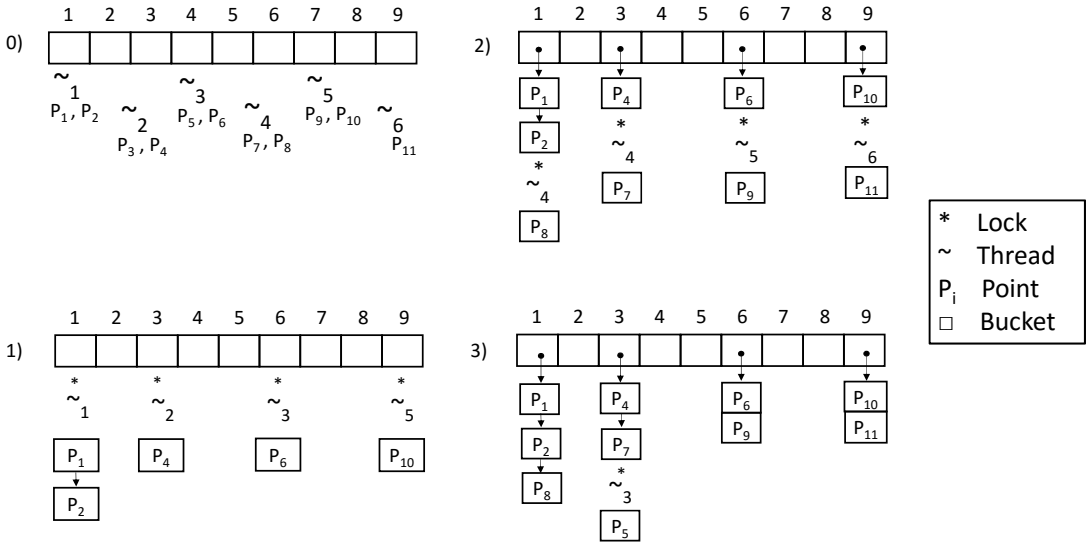


Figure 4.4: Naive bucket wise synchronisation locks to prevent different threads from inserting into the same bucket at a time, resulting in correct but inefficient insertion.

the number of points in the model and the amount of buckets used.

Creating the spatial grid using the GPU is an important step in the collision detection process, since we are assuming that this data structure can change in time and that these changes are produced on the GPU (running a physical interactive model). Since the costs of transferring information to/from the CPU memory can be prohibitive, having all the information on the GPU is advantageous. The timing of the generation of the uniform grid can be found in Table 4.1.

As can be seen in Table 4.1, when we compare these results with Table 4.1, it is clear that the GPU outperforms the CPU in most of the cases. Although the collision detection

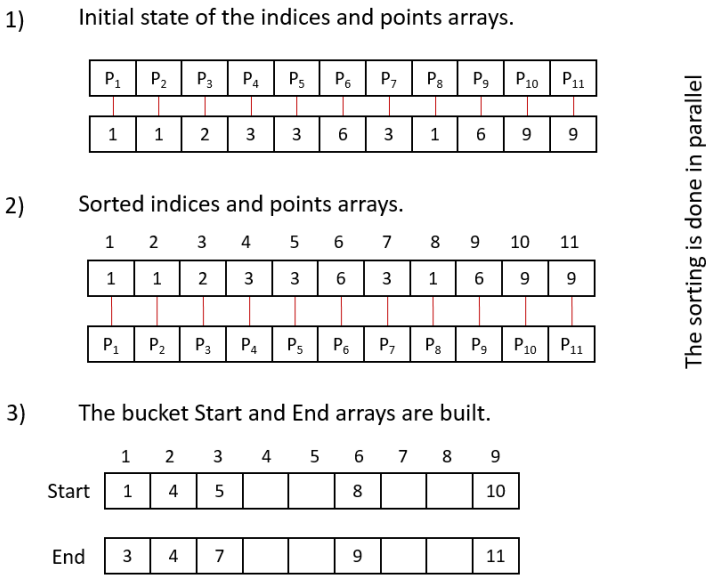


Figure 4.5: A sorting based method to insert the values into the buckets. No synchronisation is needed, the result is correct and efficient.

is faster on the CPU once the data structure has been generated, the GPU still provides response times that make haptic interaction possible using only the graphics accelerator.

These results are important since they are an order of magnitude faster than using the CPU for the same purpose and demonstrate the viability of using the GPU as the main computing platform for haptic interaction.

Although there are several other techniques for spatial partitioning such as the ones described in Section 4.1, this work implements the uniform grid because it is one of the simplest structures to generate in a massively parallel architecture, reducing the computational complexity and achieving high performance. Newer techniques [55] require more steps to compute the data structure and a more complicated querying which in turn increases the computing times for the data structure, time which in a haptic interaction application is very necessary because of the 1 ms frame time limit.

Implementing the spatial structure on the GPU is advantageous because it allows parallel processing of large point clouds very fast, as depicted in Table 4.1. Allowing the real time construction of the spatial data structure, permits the querying of the point data to determine the points within a volume and so the approximation of the surface they define. This is also important in the case the point data is morphing over time since it allows real time interaction with deformable models.

Even though the advantages of realising the proximity detection on the GPU are interesting in terms of dynamic models, they are far more complicated than what would be necessary for static models. The argument against using the GPU to generate the data structure and then processing the query on the CPU is that the time needed to transfer the data to and from the GPU can be more than what the haptic device allows. Another disadvantage is the higher complexity of the development, since although GPU programming is becoming mainstream, it still requires a high level of expertise not only on parallel algorithms but also parallel architectures, to understand required trade-offs during the design phase.

4.3.1 Brute force

Executing one thread per point in the cloud is what we been referring to as the brute force approach. Given the processing power of the GPU it is possible to achieve haptic rates using this approach for massive point clouds bigger than one million points. The drawbacks of this approach is that if the number of points in the point cloud exceeds the maximum amount of threads the GPU can execute, the points have to be batch processed degrading the speed of the haptic rendering.

4.3.2 Two Step Query

In order to have a more efficient usage of the GPU, a two step approach was developed which uses the spatial partition to select the points belonging to the buckets that are inside the interaction range of the haptic proxy. After selection the amount of points that each bucket contains is counted and the maximum number of points in the interaction buckets is determined, in order to calculate the number of threads that need to be executed to cover the maximum amount of points that can be interacting during this collision detection iteration.

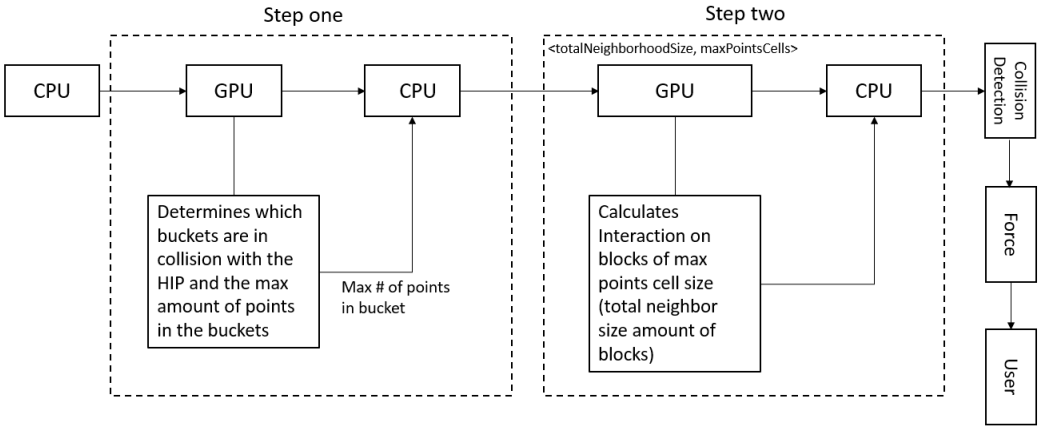


Figure 4.6: Two step method control flow diagram.

The bucket count is directly used as our block count for the kernel launch and the number of threads per bucket is the maximum amount of points in the buckets. This leads to a sub utilization of the GPU which is addressed in the following sections using the dynamic parallelism technology.

4.3.3 Dynamic Parallelism

Dynamic parallelism extends the idea of executing a set of operations in parallel on the GPU with an initial configuration of the grid of blocks mentioned in Section 2.4.1. Using dynamic parallelism the grid can be configured on the GPU without the need to return control to the host to determine the launch configuration. Using dynamic parallelism a thread can invoke a new grid and takes control of all the threads invoked, allowing for an implicit synchronization. Figure 4.7 depicts an image of the control flow followed by a dynamic parallelism enabled kernel launch.

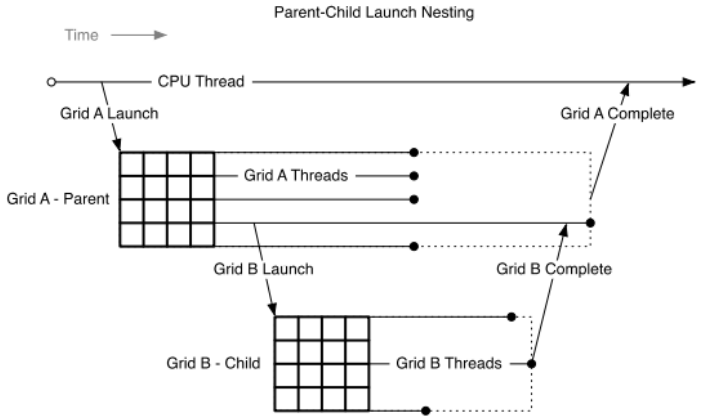


Figure 4.7: Control flow of parent grid to thread child grid on the device over time. [3]

An important feature of dynamic parallelism is that it allows a kernel to have program flow control. Thus, reducing PCI traffic if the host needed information from the device in order to configure a kernel launch. Dynamic parallelism also allows hierarchical algorithms to be expressed in a more straight forward way. Using dynamic parallelism different levels of detail for the execution of an algorithm can be resolved using a single initial kernel launch from the host and then having multiple sub-launches configured on the device from the initial one. Therefore, an initially coarse grid can be improved to a finer grid on the places where it is necessary, improving the efficiency on the usage of the GPU computing power.

In order to utilize as efficiently as possible the GPU computing power available on cards capable of dynamic parallelism it is necessary to implement hierarchically based search queries to identify the points in the model that the user is interacting with. In the following sections, a dynamic parallelism version of the spatial queries are described.

Naive implementation

In order to fully utilize the computing power of the GPU and avoid wasting computing resources on areas of the model that are far from the HIP. Dynamic parallelism provides a way to allocate computing resources to the region of interest and dynamically invoke threads to process the points within that region. This is important since it helps minimize the overhead of data transfer times through the PCI bus and the synchronization time required between the CPU and the GPU. In this section an implementation of range search around the HIP based on dynamic parallelism is presented.

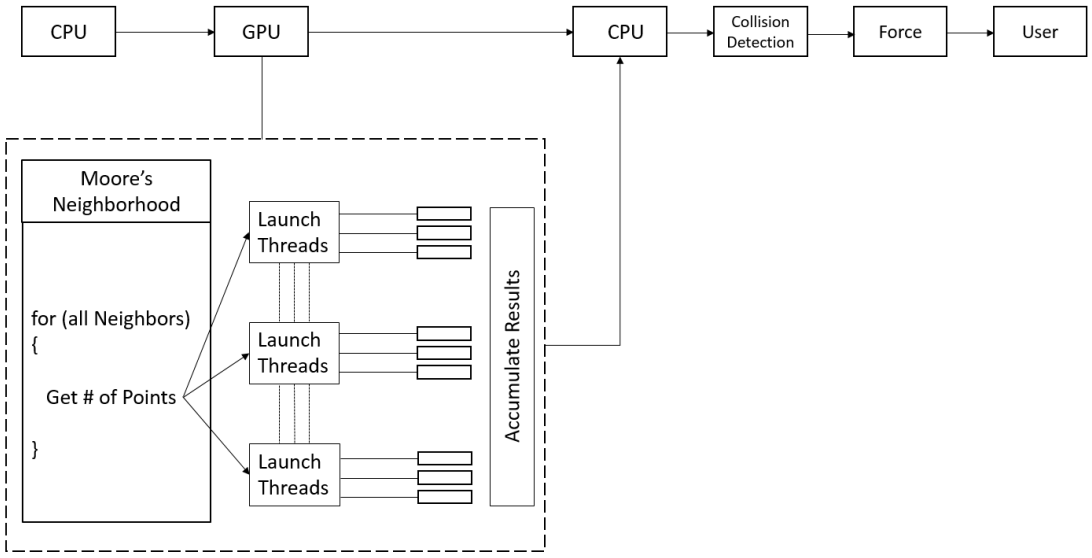


Figure 4.8: Control flow of the Naive implementation using dynamic parallelism.

The algorithm is very straightforward, initially the host launches a single thread on a

single block that will be responsible for cycling through the neighbouring buckets within the region of interest based on the size of the influence radius defined by the user. Then, for each bucket a new grid is launched with a number of threads equal to the quantity of points in the bucket. Finally, each child thread on the device calculates the interactions between the HIP and the model.

This method is more efficient than previous ones since it avoids transfers between the host and the device which can slow down the query and it does not require every point in the model to be tested like the method described in Section 4.3.1.

Array based implementation

This version of the dynamic parallelism implementation reduces the number of kernel invocations required by the approach from Section 4.3.3. Instead of launching the kernels as the buckets are identified, it loads the indices of the vertices in the buckets on a temporary array. With a length equal to the maximum number of points that can be interacting on any iteration, this quantity is estimated using the number of neighbouring buckets and the points per bucket.

The main advantage of this method is the drastic reduction in kernel invocation from the main thread, gaining performance since the overhead of configuring and launching child kernels is reduced.

The main disadvantage of this method is the memory footprint required by it, since the storage necessary is difficult to determine prior to the kernel launch an estimation needs to be used, leading to unnecessary memory being reserved for this algorithm.

Looking at Figure 4.8 and comparing with the graphical control flow of the array based method from Figure 4.9 we can easily visualize the difference in execution, where the naive

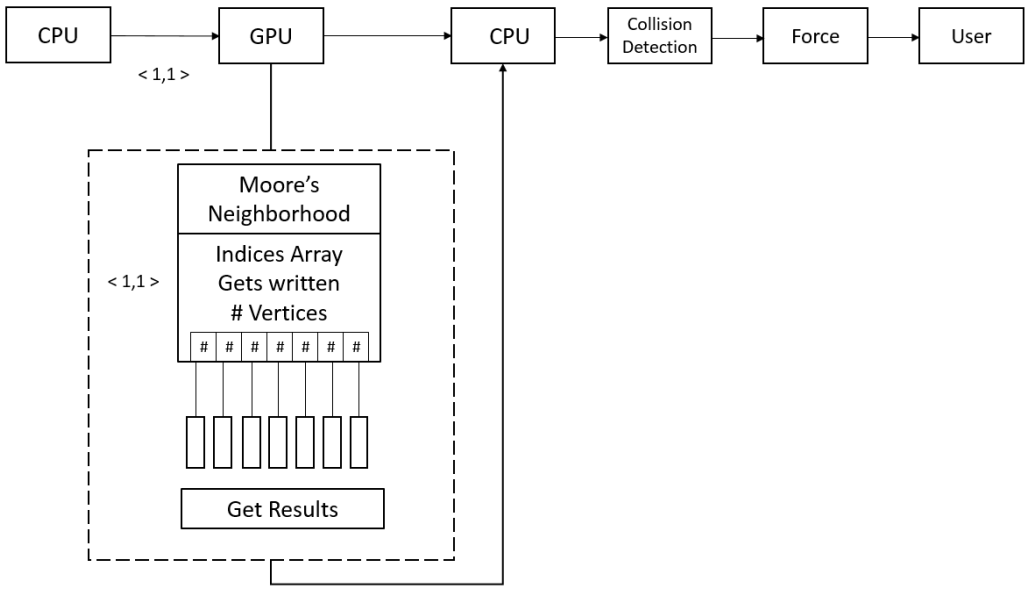


Figure 4.9: Control flow of the array based query.

version launches a large number of sub-kernels, the array based approach just prepares the configuration to make a single optimal kernel launch mapping each computing grid block to a single bucket in the spatial subdivision structure. This is the key difference between the methods and provides an improvement in the execution timings because the array based version minimizes instruction divergence and kernel invocation overheads.

Finally, dynamic parallelism provides a tool for interacting with very large models in an efficient way since the GPU only does the minimum required work while calculating the HIP to model interactions.

4.4 Results

In this chapter we have described a CPU and a GPU based implementation of the spatial hashing technique. These techniques differ not only on implementation but also require

different amounts of time for construction. Here we present an overview of the different timings for the construction of the CPU based spatial hashing technique. The results for the querying are shown in the next chapter since they are reported with the haptic rendering times.

The method utilized to measure and compare the timings for the different implementations is the following: First three models with varying numbers of points were selected, a generated cube model, the Stanford bunny model, the Dragon, the Happy Buddha and the Armadillo model along with the point clouds of two chest pieces from a Cantonese chess set and the point cloud of a rock model obtained using photogrammetry. These models have 107409, 208347, 437645, 543652, 598562, 819951 and 1037774 points respectively. The timing measurements do not include any transfer times or model reading times. Only the insertion of the points in the data structure is measured.

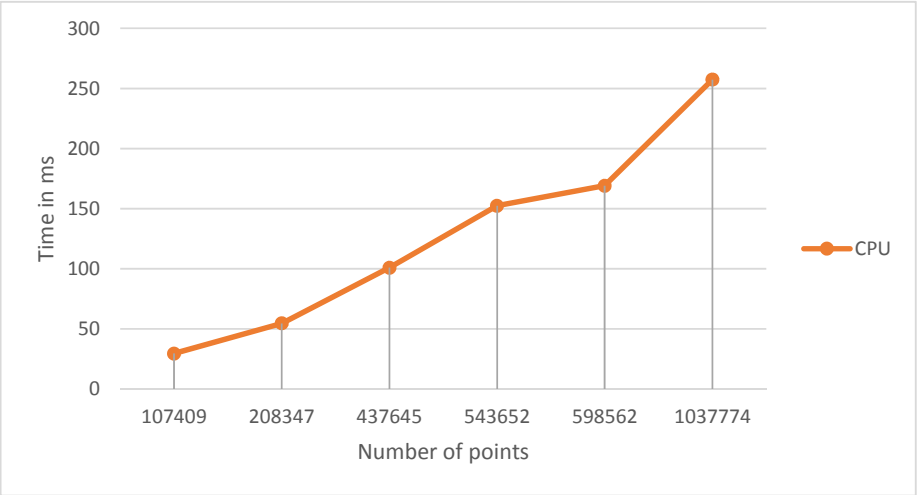
The proposed method was implemented in visual C++ in windows 10 platform. To measure the performance of the method the hardware used had following characteristics: Processor Intel(R) Core(TM) i7 CPU 4720HQ @ 2.60 GHz with 8.0 GB of RAM, and a GeForce GTX 270M with 3.00 GB of RAM memory.

Model	# Points	CPU (ms)	GPU (ms)
Cube	107409	28.62	2.04
Bunny	208347	53.36	2.54
Dragon	437645	100.26	3.89
Happy	543652	147.24	4.54
Archer	598562	178.99	4.63
Rock	819951	243.40	5.39
Armadillo	1037774	255.20	9.22

Table 4.1: Hash table construction timings measured in milliseconds using an Intel(R) Core(TM) i7 CPU 4720HQ @ 2.60 GHz and a GeForce GTX 270M.

Comparing the results between the CPU and the GPU implementation we can observe from Figure 4.10 that the timing trends look similar on both implementations, but looking

at the data in Table 4.1 we can easily see that on the CPU implementation the scaling is almost linear with respect to the amount of points at a factor averaging 3784 points per ms, whereas the GPU implementation scales at a different pace, having a factor of 48648 points per ms for the first model, 80381 points per ms for the second model, 112530 points per ms for the third model, 119703 for the fourth model, 129353 for the fifth model, 152002 for the sixth model and 106489 for the seventh model. Compared to the CPU implementation the GPU achieved a speedup of about 20 times the CPU version while constructing the data structure. Achieving a speed within haptic interaction constraints for the construction of the data structures.



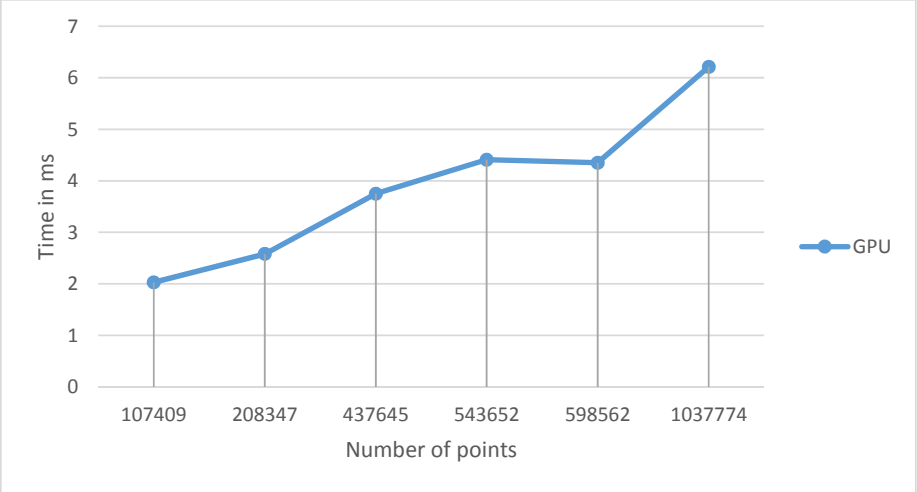


Figure 4.10: Spatial subdivision structure construction time on the CPU and GPU measured in milliseconds, using all the models shown in Table 4.1.

5 Haptic Rendering

The possibility of interacting with virtual objects in the computer simulation already exists for sight and hearing, it is only relatively recently that the sense of touch is being introduced to the human computer interface. The process through which the user interacts with the computer via the sense of touch is called haptic rendering.

Haptics enhance the interaction between humans and computers by including one more human input/output system. There are several advantages of using the sense of touch while interacting with computer systems, some of these are the increased concentration that can be achieved by the user during the immersive experience, also the possibility of interacting with virtual environments in a more representative way such as being able to feel, move or change objects in the virtual space just like someone would in the real world.

On the other hand, including haptics in a virtual environment can be a difficult task because of the strict requirements for realistic haptic interaction. These requirements include a very fast rate of computation as has been discussed in Section 2.6.1, this rate of computation is necessary for a satisfactory user experience. Also, the quality of the interaction depends on the level of detail of the models used to represent the forces exerted by the haptic device to the user, these forces are obtained directly from the results of the collision detection algorithms that need to be fast enough to provide a quick query of the space so the rendering system can calculate an adequate force to apply

to the user in response to haptic input.

In our work, we use constraint-based haptic rendering for the haptic feedback. A penalty force is calculated based on the distance between the probe and the proxy point. This method is commonly used for surface interaction. The surface definition based on the 3D point cloud is calculated using a weighted contribution of the distance between the proxy and the point samples.

Specifically, the model used for the force rendering can be expressed as:

$$F = k_{spring} * \|\triangle X\| + k_{damper} * \|\triangle \dot{X}\| \tag{5.1}$$

Where k_{spring} is the spring constant in Hooke's law directly connected to the force the user would have to apply to increase the distance between the HIP and the SCP. A larger value for this parameter would increase the perceived hardness of the model, while a lower parameter would make the surface feel softer. The term $\|\triangle X\|$ defines the difference between the position of the HIP and the SCP. Finally, k_{damper} is the damping coefficient which smooths abrupt changes in the force using the rate of change in the distance between the HIP and SCP represented by the term $\|\triangle \dot{X}\|$.

And the method used to represent the implicit surface that provides the basis for the constraints necessary in the god-object technique based on Leeper, Chan and Salisbury 2012 [33] is:

$$\begin{aligned} \mathbf{a}(\mathbf{x}) &= \left(\sum_i w_i \mathbf{p}_i\right) / \left(\sum_i w_i\right) \\ \mathbf{n}(\mathbf{x}) &= \left(\sum_i w_i \mathbf{n}_i\right) / \left(\|\sum_i w_i \mathbf{n}_i\|\right) \end{aligned} \tag{5.2}$$

Where for any given query location $\mathbf{x}$, we define the weighted average $\mathbf{a}(\mathbf{x})$ of the point positions $\mathbf{p}_i$, and the weighted average $\mathbf{n}(\mathbf{x})$ of the point normals.

Then the implicit equation and gradient are:

$$\begin{aligned} f(x) &= \mathbf{n}(\mathbf{x})^T(\mathbf{x} - \mathbf{a}(\mathbf{x})) \\ \nabla f(x) &\approx \mathbf{n}(\mathbf{x}) \end{aligned} \tag{5.3}$$

$f(x)$ defines the distance a given point lies on either side of the surface approximation defined by $\mathbf{a}(\mathbf{x})$ along the normal direction. The definition of the surface and the surface normal defined by Equation (5.3) are used to place the proxy on the surface of the object and to give direction to the force exerted by the haptic device, the orientation and placement of the disk and vector evidenced in Figure 5.1 is the graphical representation of the application of this equation.

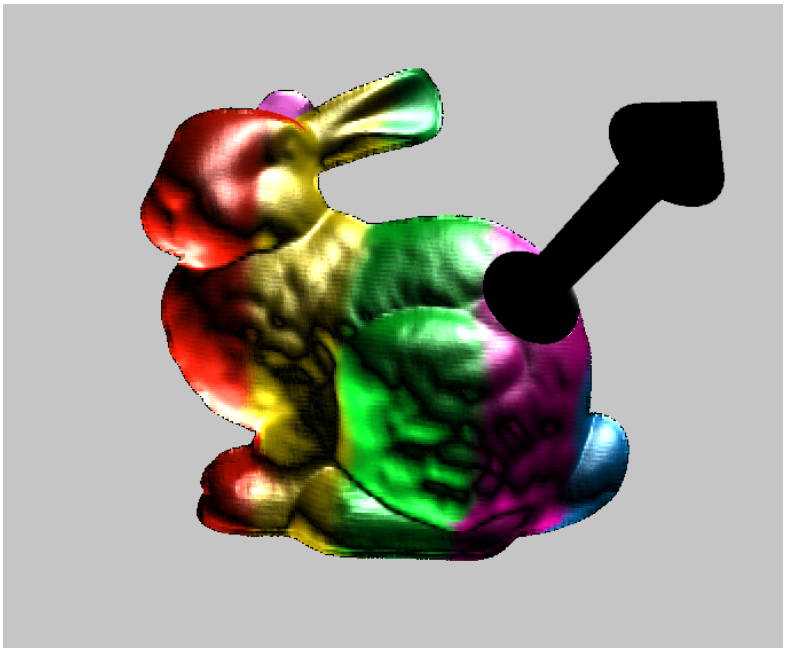


Figure 5.1: The disk is positioned on top of the surface of the model as a result of the haptic rendering algorithm. The force vector is oriented by the normal of the surface.

5.1 Static model single point rendering

The method used for our single point haptic rendering of a point model is inspired by Salisbury 1997, [48]. To find the nearest point on the surface of our model given our interaction point we use the following algorithm:

Algorithm 6 Finding a point on the surface S of a model given a point p

1: $\mathbf{p} \leftarrow \mathbf{p}_{seed\_point}$
2: **do**
3: $\quad \delta\mathbf{p} \leftarrow -\frac{S(\mathbf{p})\nabla S(\mathbf{p})}{\nabla S(\mathbf{p})\cdot\nabla S(\mathbf{p})}$
4: $\quad \mathbf{p} \leftarrow \mathbf{p} + \delta\mathbf{p}$
5: **while** $\|\delta\mathbf{p}\| > \epsilon$
6: $\mathbf{p}_{surface\_point} \leftarrow \mathbf{p}$

The timing results for a static 3D point cloud model with a single point rendering method are presented in this chapter, along with the configuration of the scene and the description of the hardware where the timings were measured.

The scenes are based on point data obtained from 3D models by the Michelangelo project [35], simple Blender generated geometric figures and a 3D point-based reconstruction of a rock. A 3D modelled cube, the Bunny, Buddha, Dragon and Armadillo models obtained from the Michaelangelo project, a 3D point cloud from a CT Scan of a Cantonese chess piece and the point-based reconstruction of a rock's surface are the test models.

The algorithm in the case of the static single point rendering case can be described in the following steps:

Load PLY file The model is read from a ASCII or binary format PLY file to memory.

Create Spatial Subdivision Structure The technique for spatial hashing (SH) described in Section 4.1 is generated either on the GPU or CPU.

Initialize interaction device The device to interact with the model is initialized, can be

a haptic device or a keyboard/mouse simulated haptic device when there is no haptic device present.

Query the SH The position of the device is updated and used to query the SH for interaction with the model.

Calculate implicit surface around HIP Using the HIP or the proxy in case the device is in contact with the surface. Calculate the implicit surface around the query point and return the results of $\mathbf{a}(\mathbf{x})$ and $\mathbf{n}(\mathbf{x})$ back to the CPU.

Use $\mathbf{a}(\mathbf{x})$ and $\mathbf{n}(\mathbf{x})$ to update the proxy position Using Algorithm 6 we update the proxy position.

Output forces Output the force to the user using Equation (5.1) and the vector between the HIP and the Proxy.

A graphical representation of the process described in this section can be found in Figure 5.2.

5.2 Deformable model single point rendering

Applying deformations to the model while doing haptic interaction is a challenging task especially since in this work there is no intermediate representation of the model and the models we are working with are of at least 100000 points which makes them more difficult to work with since the spatial subdivision structure needs to be updated as the model is modified.

A description of the process the model goes through during interaction is described in the following lines:

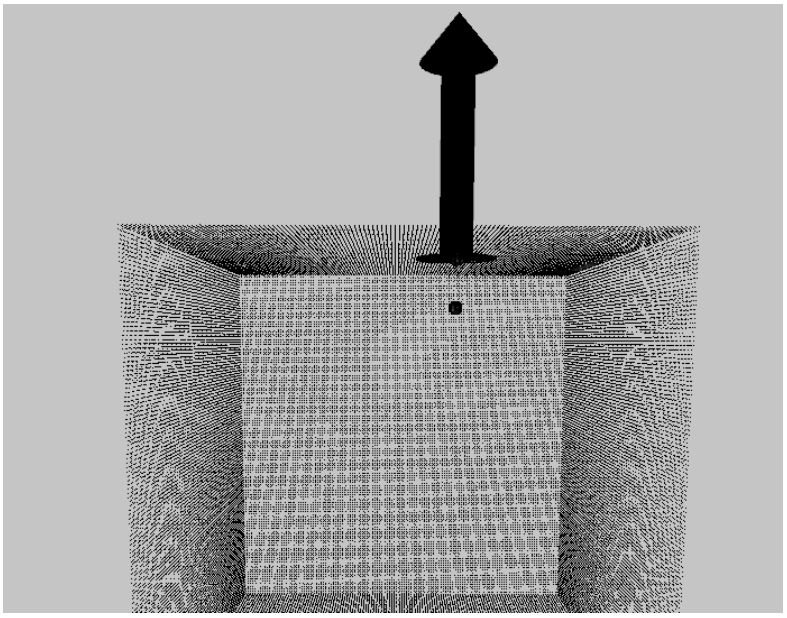


Figure 5.2: The proxy object is oriented by the surface of the cube, the vector represents the distance between the HIP and the proxy. The HIP is represented by a sphere found inside the object below the proxy.

Load PLY file The model is read from a ASCII or binary format PLY file to memory.

Create Spatial Subdivision Structure The technique for spatial hashing (SH) described in Section 4.1 is generated either on the GPU or CPU.

Initialize interaction device The device to interact with the model is initialized, can be a haptic device or a keyboard/mouse simulated haptic device when there is no haptic device present.

Query the SH The position of the device is updated and used to query the SH for interaction with the model.

Calculate implicit surface around HIP Using the HIP or the proxy in case the device is in contact with the surface. Calculate the implicit surface around the query point and return the results of $\mathbf{a}(\mathbf{x})$ and $\mathbf{n}(\mathbf{x})$ back to the CPU.

Use $\mathbf{a}(\mathbf{x})$ and $\mathbf{n}(\mathbf{x})$ to update the proxy position Using Algorithm 6 we update the proxy position.

Check distance magnitude If the distance between the HIP and the Proxy is larger than τ store in the force array $f_i = (Proxy - P_i) * \beta$.

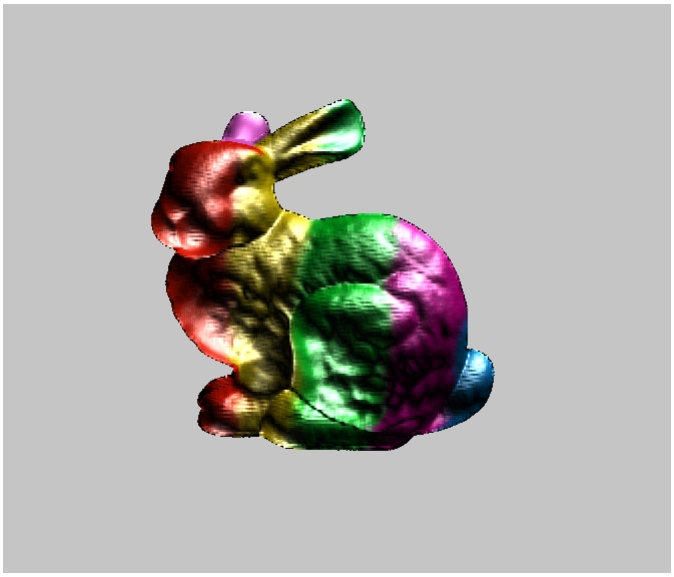
Mark model for update Mark the model's points positions as *dirty* and to update the spatial structure in the next frame.

Output forces Output the force to the user using Equation (5.1) and the vector between the HIP and the Proxy.

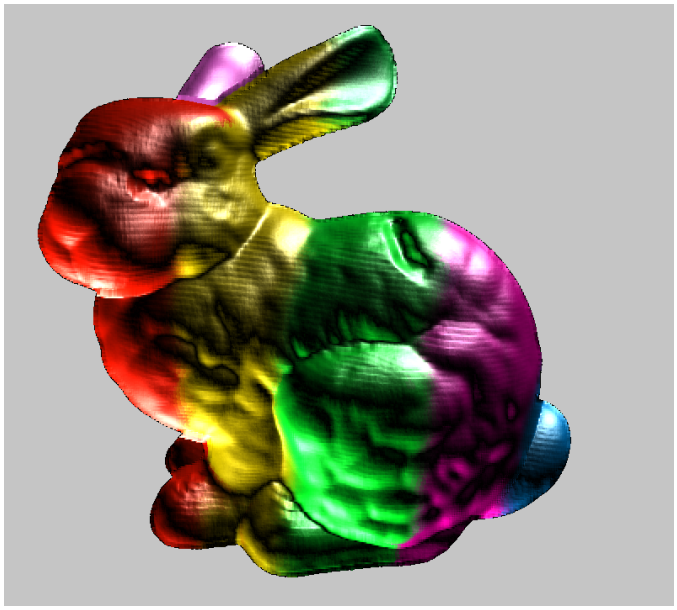
Update the positions Using the force array update the positions of the points in the model as $\mathbf{p}_i = \mathbf{p}_i + f_i$.

Update the spatial structure Once the forces have been applied update the models points positions and clean the *dirty* flag.

We chose the values of τ and β empirically to maximize the realism of feeling a solid surface model being poked with a round point tool. Figure 5.3 illustrates the result of the interaction using the deformable model single point rendering algorithm.



(a)



(b)

Figure 5.3: The difference between an unmodified version of the bunny model and a modified one. The image in the bottom depicts the surface of the model after being edited using the haptic tool, an indentation can be observed along the top right of the model and the face.

The main strength of the implementation described in this section for highly detailed

models, where the spatial data structure is continuously being updated or more precisely recreated, is that the implementation can handle the case of continuously changing models as can be seen in Section 5.3. For example, a camera mounted on top of a robot providing haptic feedback in real-time, a similar implementation can be found in Kaluschke et al [29]. Instead of an Inner Sphere Tree (IST) we only need single point collision detection and we define the surface of the model for user interaction.

5.3 Results

The proposed method was implemented in visual C++ in windows 7 platform. We used 3 hardware configurations for this project A, B and C. With the following characteristics:

A Processor Intel(R) Core(TM) i7 CPU 870 @ 2.93 GHz with 16.0 GB of RAM, and a NVIDIA GeForce GTX 480 with 1.50 GB of RAM memory.

B Processor Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40 GHz with 4.0 GB of RAM, and a NVIDIA GeForce GTX 780 with 1.00 GB of RAM memory.

C Processor Intel(R) Core(TM) i7 CPU 4720HQ @ 2.60 GHz with 8.0 GB of RAM, and a GeForce GTX 270M with 3.00 GB of RAM memory.

The spatial data structure is calculated using a regular grid of 128x128x128. We use the 3-DOF Phantom Omni haptic device from SensAble, only available for machine A. Moreover, the object is visually rendered using the technique described in Section 3.3 at a fixed frame rate of 30 fps. This allows the user to have visual and haptic feedback while interacting with the object, improving the feeling of realism and immersion.

In Table 5.1 we can observe the measurements for the Armadillo model using an influence radius big enough to include 3 buckets around the HIP on each dimension.

Model	# Points	Hardware	Brute	Brute+	Naive	Naive+	Dyn	Dyn+
Armadillo	1037774	A	0.97	1.49	N/A	N/A	N/A	N/A
Armadillo	1037774	B	0.91	3.03	1.43	3.00	0.83	2.98
Armadillo	1037774	C	0.91	3.78	1.00	3.18	0.54	3.20

Table 5.1: The hardware used and the timings in ms for the different methods implemented for spatial querying including the force rendering (Brute force, Naive and Array based), the + sign indicates the model is being edited during interaction.

This influence radius is much bigger than needed since we are using dense point cloud models and the points required for the correct rendering of the forces are smaller, but is useful to test the limits of the methods. The timings where measured for three of the querying methods described in Section 4.3. These timings include calculating the implicit surface and the haptic force rendering described in Algorithm 6, the overhead for computing both the haptic force and implicit surface are negligible since they are simply a few vector-vector and vector-scalar operations.

In the table the last two methods timings are not available for hardware A because that GPU does not support dynamic parallelism. Using different hardware configurations was important to analyse the performance scaling. Particularly, the difference resulting from the usage of newer capabilities in more recent graphics cards such as dynamic parallelism.

Now we will discuss the results for each hardware, how the different methods behave while querying and rendering forces for the models on each architecture.

Hardware A

On hardware A only the bruteforce and two-step methods can be evaluated because the compute capability of this graphics card does not support dynamic parallelism, which is necessary for the naive and array based approaches. It can easily be seen how the timings of the methods deteriorate as the models grow in number of points, these timings

Model	# Points	brute	brute+	2step	2step+	naive	naive+	dyn	dyn+
Cube	107409	0.23	0.29	0.45	0.56	-	-	-	-
Bunny	208347	0.30	0.40	0.56	0.71	-	-	-	-
Dragon	437645	0.42	0.56	0.67	0.83	-	-	-	-
Happy	543652	0.56	0.67	0.56	0.71	-	-	-	-
Archer	598562	0.63	0.83	1.43	1.67	-	-	-	-
Rock	819951	1.25	2.50	2.00	2.22	-	-	-	-
Armadillo	1037774	0.97	1.49	1.25	2.50	-	-	-	-

Table 5.2: Timings for the methods, using hardware **A**. The times are measured in ms, the + sign indicates the model is being edited during interaction.

even suggest that the two-step approach is not ideal on this particular card since it only provides worse performing timings, only for the Happy Buddha model the two-step approach achieves the same speed as the bruteforce.

Hardware B

Model	# Points	brute	brute+	2step	2step+	naive	naive+	dyn	dyn+
Cube	107409	0.37	0.42	1.00	1.25	1.43	1.67	0.59	0.63
Bunny	208347	0.42	0.77	1.25	1.27	1.11	1.61	0.56	0.91
Dragon	437645	0.56	1.35	1.64	2.22	1.25	2.13	0.63	1.43
Happy	543652	0.63	1.61	1.12	2.13	1.43	2.50	0.77	1.75
Archer	598562	0.67	1.79	2.17	2.56	1.56	3.33	1.00	2.13
Rock	819951	0.83	2.94	1.82	3.57	1.45	3.45	0.91	2.94
Armadillo	1037774	0.91	3.03	1.89	3.33	1.43	3.45	0.83	2.98

Table 5.3: Timings for the methods, using hardware **B**. The times are measured in ms, the + sign indicates the model is being edited during interaction.

The results of the timings for the different methods using hardware B can be found in Table 5.3. There we can observe how the bruteforce and two-step approaches deteriorate as the number of points increase although the two-step approach reaches minimum at approx. 2 ms while the bruteforce decreases up to approx. 3.33 ms, from the implementation it can be estimated that the bruteforce approach timings would continue to decrease as the number of points tends to infinity, while the two-step method reaches a minimum and

stabilizes.

Different from the bruteforce and two-step methods affected by the number of points present in the model, the dynamic parallelism enabled methods, much higher minimum rates with the naive approach falling to approx. 1.43 ms and the array based approach bottoming at approx. 1.1 ms. The timings for these methods when the editing flag is set is dominated by the spatial structure reconstruction.

Hardware C

Model	# Points	brute	brute+	2step	2step+	naive	naive+	dyn	dyn+
Cube	107409	0.25	0.50	0.53	0.71	1.15	1.54	0.42	0.65
Bunny	208347	0.32	0.91	0.63	1.02	0.91	1.43	0.40	1.00
Dragon	437645	0.57	2.04	0.67	2.00	1.11	2.56	0.50	2.00
Happy	543652	0.59	2.27	1.25	2.33	1.59	3.23	0.71	2.26
Archer	598562	0.48	1.67	0.56	1.67	1.18	2.34	0.43	1.67
Rock	819951	0.71	3.33	0.63	3.70	0.92	3.75	0.51	3.33
Armadillo	1037774	0.91	4.35	1.37	3.47	1.00	4.00	0.54	3.45

Table 5.4: Timings for the methods, using hardware **C**. The times are measured in ms, the + sign indicates the model is being edited during interaction.

The newest hardware available for tests is C, with a GeForce GTX 270M that has compute capability 5.2 (the latest available for NVIDIA Corporation), the timing results for this hardware are shown in Table 5.4. Similar to the previously presented hardware A and B, the bruteforce and two-step methods decrease in performance as the number of points increases in the models.

The timings for two other Stanford 3D Scanning Repository models where measured using this hardware. To test the upper limits of the methods, the Asian Dragon and the Thai Statue models, containing 3609455 and 4999996 points where used. The haptic rendering results obtained for these models confirm the trends presented in Table 5.4. The bruteforce method only achieves 3.33 ms for the Asian Dragon and 4 ms for the

Thai Statue, whereas the array based method, achieves haptic rendering rates of 0.71 ms and 1.1 ms respectively.

Hardware C presents similar results to hardware B in terms of scaling respect to the number of points, but the newer graphics card in C presents an improvement averaging 18% for the naive query implementation and 31% for the array based approach. These improvements suggest that the methods for space querying based on dynamic parallelism described in this thesis scale well with newer architectures.

5.4 Discussion

Overall the differences between the methods can be easily observed in Table 5.1, displaying the scaling of the different methods for the largest of the models available, the results show that the dynamic parallelism based methods provide a strong performance improvement over the brute force approach, working up to three times faster as is the case for hardware C timings.

Also, comparing the results from Tables 5.3 and 5.4, we observe how the newer architectures besides providing the more recent dynamic parallelism functionality, have improved the timings of invoking kernels within a thread by reducing the overhead of the invocation logic, this result can be obtained from comparing the naive implementation to the array based between hardware B and C.

The haptic rendering technique presented in this thesis is constrained by the memory of the GPU in the case of static models. Particularly, the limiting factor based on the size of the GPU RAM, the amount of memory required by a model is $2 * nPoints * 12$ bytes for the vertex information (position, colour and normal), plus $2 * nPoints * 4$ bytes for the indices array, plus $2 * hashSize * 4$ bytes for the bucket's *start* and *end* arrays.

Also, a memory buffer is required for the array based version of the neighbour search, this buffer has a size of $4(2r+1)^3$ bytes where r represents the maximum size measured in buckets of the search radius.

As the number of points increases, the brute force technique starts to fall off in performance, this is given to the exhaustive nature of the neighbour search represented by this method. On the other hand, the naive and array-based methods employ a spatial locality search taking advantage of the spatial structure, limiting the search of the neighbouring points and achieving very stable rendering rates regardless of the number of points present in the model. The bigger models available in the Stanford repository of three million and five million points can be rendered by these techniques at haptic rates, demonstrating their validity.

We have found that at the one million point mark, the edit enabled methods start to perform on the lower limits of haptic rates. Hence, the model editing methods presented in this work are limited by the spatial structure construction times. The scalability of the methods suggests that increasing the number of points beyond the one million mark would require faster hardware or the development of new techniques.

Figure 5.4: The visual-haptic system: the user explores the haptic space. He feels a countering force when pushing against the surface of the model.

6 Conclusions

Presenting complex geometry and material properties is an area of interest in the computing sciences, more specifically involving the user in a realistic environment that is completely simulated by various mathematical models. Among the models that can be presented to the user of a computing system of virtual reality, the sense of touch is of special interest since it helps the user have a deeper experience of the world that surrounds him and a better ability to interact with it, given that from an early stage of development the human being is constantly touching and grasping objects to form an understanding of the world. Not only the haptic feedback is important for a convincing virtual environment but also the visual feedback that the user receives plays a crucial role at improving the immersive experience. In this thesis techniques have been developed for both of the senses mentioned to create a foundation for different applications that require visual and haptic feedback, such as 3D modelling tools or immersive robotics control applications for hapto-visual environment exploration.

The haptic rendering system is complex and involves many parts acting together, one important part of this system is the human user, who, along with the computer and haptic device complete the necessary elements for a simulated virtual environment. Regardless of what algorithm or method is used, the main focus of the study is to develop a combination of techniques and models that can generate a realistic force feedback, in order to augment the perception of realism.

Most of the existing interactive systems focus primarily on the audiovisual aspect of virtual environments, but recently other senses are being involved and research is focusing on different interaction pathways such as the sense of touch, in this sense, haptic interaction is gaining popularity and has found a place in virtual reality systems, being one of the main applications mainly focused on medical and mechanical virtual environments. Haptics accompanied by a quality graphical rendering method can produce systems advantageous to a variety of domains such as industrial design, three-dimensional navigation, art and robotics.

This chapter summarises the contributions, limitations and challenges of the methods presented. An overview of the general algorithm followed by the system to generate haptic and visual cues to the user about the virtual environment is initially presented, followed by a discussion of the graphic and haptic rendering processes.

6.1 System overview

The focus of this study has been developing techniques to enable three-dimensional haptic interaction with high detailed point based models. A big challenge of this study was the fact that the models contained a high number of primitives that need to be tested for proximity with the haptic tool, this challenge was tackled by the usage of graphic processing units as a coprocessor helping the system accelerate the collision detection and fulfil objective 1a.

Currently, most haptic interaction systems are based on static models or models with a limited representation of reality such as low detail or approximations. This work focuses on working with the originally provided model and few approximations to increase realism needed for applications in the medical, engineering and creative processes. The datasets used in this work contained up to one million points, in accordance with objective 1b.

The techniques presented in this thesis demonstrate sufficient evidence that the usage of the GPU can be effective within the haptic interaction field, admitting the high overhead paid by the GPU to CPU communication. The methods described in the thesis focus on constructing a direct graphical representation for the point based models and a haptic rendering technique aided by the GPU to achieve the necessary haptic rates with high detailed models. The construction of a system capable of representing and modifying a high detail point based model haptically, along with its visual representation, was successfully accomplished by this work satisfying objective 1c and 2.

6.2 Graphic & Haptic rendering

The graphics low rendering times exhibited by the screen space technique presented in this work along with the low memory footprint represent an interesting alternative to polygon based rendering methods. Using screen space techniques a continuous high quality surface of a point based model can be achieved. A small number of parameters are used to adjust the quality of the graphical representation of the model, these parameters can increase or decrease the rendering times depending on the number of pixels that need to be processed. Given that both haptic and graphics rendering were using the GPU, the graphics rendering parameters affected the haptic rendering times, because of this, the graphics display rate was fixed to 30 frames per second in order to limit the GPU resources used for graphics rendering. Limiting the resources used by the graphics allowed the GPU to provide more consistent and faster haptic rendering times.

The research presented in this thesis regarding haptic interaction, concentrates on rendering implicit surfaces originated from 3D point cloud models. The methods described in the thesis achieve haptic rates while interacting with high detailed models consisting of several hundred thousands of points. Equally, editing of the point clouds is achieved

by the methods presented in this thesis.

Finally, the main result of this work is the implementation haptic interaction system that renders high detailed point based models graphically and haptically while also allowing the user to edit and modify the model during interaction. This can be the groundwork for a more complex haptical rendering tool for point based modelling, haptic surface exploration and similar applications where the information that is being interacted with changes quickly over time.

6.3 Limitations & Future work

This thesis focuses in the methods to provide haptic interaction with high detailed point based models. However, there is no physical simulation model behind the interactions when modifying the point model. Any physical model that could be incorporated would need to be implemented on the coprocessor (the GPU in this case) and then connected with the surface represented by the point cloud.

Since the data modification model is not based in physical properties, this can be an area of improvement and future work. Several alternatives can be studied to make a more realistic dynamic model of the point clouds, such as active contour based models where the surface is defined by an energy function that could be affected by the haptic device incorporating a plasticity term in the formulation of the contour energy, a finite element method with a lower detailed model or a mass-spring system connected to the point cloud using a skinning approach.

During the study a general problem was identified, the haptic rendering of high curvature areas on implicit surfaces is still weakness within the haptic rendering methods available, since no technique has proven to be effective while dealing with these. Improving

the quality of haptic rendering of high curvature areas defined by implicit functions would be beneficial for a wide range of haptic rendering applications.

The solution to the problems previously stated was not explored since it would require a major effort in writing GPU versions of the algorithms, since the point cloud is entirely on the GPU so the transfer times between the CPU and GPU are prohibitive.

It would also be interesting to study the possibility of partially updating the spatial data structures, by implementing a new method for its creation which allows easier updates or by studying the possibility of improving the GPU sorting algorithms for partially sorted arrays. The methods described in this thesis are based on sorting the full data set similarly to other methods present in the literature, studying how to use the GPU to sort partially ordered arrays in an efficient way might be the key that opens a new range of applications for GPUs in spatial querying, collision detection and resolution.

Bibliography

[1] Best practices guide, cuda toolkit documentation. `docs.nvidia.com/cuda/cuda-c-best-practices-guide`. Accessed: 2015-11-01.

[2] Cuda c programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#programming-model`. Accessed: 2015-11-01.

[3] Adinetz, A. CUDA Dynamic Parallelism API and Principles. Accessed: 5-12-2014.

[4] Appel, A. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of the 1967 22nd national conference on* - (New York, New York, USA, Jan. 1967), ACM Press, pp. 387–393.

[5] Barbic, J., and James, D. Six-DoF Haptic Rendering of Contact Between Geometrically Complex Reduced Deformable Models. *IEEE Transactions on Haptics 1*, 1 (Jan. 2008), 39–52.

[6] Bickley, L., and Szilagyi, P. G. *Bates' guide to physical examination and history-taking*, vol. 13. Lippincott Williams & Wilkins, 2012.

[7] Blausen Medical Communications, I. http://blausen.com/?Topic=8444 acessed on, 2014.

[8] Botsch, M., Hornung, A., Zwicker, M., and Kobbelt, L. High-quality

surface splatting on today's GPUs. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* (2005), IEEE, pp. 17–141.

[9] Botsch, M., and Kobbelt, L. High-quality point-based rendering on modern GPUs. In *11th Pacific Conference onComputer Graphics and Applications, 2003. Proceedings.* (Oct. 2003), IEEE Comput. Soc, pp. 335–343.

[10] Buchart, C., Borro, D., and Amundarain, A. GPU Local Triangulation: an interpolating surface reconstruction algorithm. *Computer Graphics Forum 27*, 3 (May 2008), 807–814.

[11] Dachsbacher, C., Vogelgsang, C., and Stamminger, M. Sequential point trees. *ACM Transactions on Graphics 22*, 3 (July 2003), 657.

[12] Dai, X., Gu, J., Cao, X., Colgate, J. E., and Tan, H. SlickFeel. In *Adjunct proceedings of the 25th annual ACM symposium on User interface software and technology - UIST Adjunct Proceedings '12* (New York, New York, USA, Oct. 2012), ACM Press, p. 21.

[13] De, S., Lim, Y.-J., Manivannan, M., and Srinivasan, M. A. Physically Realistic Virtual Surgery Using the Point-Associated Finite Field (PAFF) Approach. *Presence: Teleoperators and Virtual Environments 15*, 3 (June 2006), 294–308.

[14] Debevec, P. E., Taylor, C. J., and Malik, J. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 11–20.

[15] Dervaux, F., Peterlik, I., Dequidt, J., Cotin, S., and Duriez, C. Haptic Rendering of Interacting Dynamic Deformable Objects Simulated in Real-Time at Different Frequencies. In *IROS - IEEE/RSJ International Conference on Intelligent*

Robots and Systems (Nov. 2013).

[16] Dyken, C., and Ziegler, G. High-speed Marching Cubes using Histogram Pyramids. *Eurographics 26*, 3 (2007).

[17] El-Far, N. R., Georganas, N. D., and El Saddik, A. An algorithm for haptically rendering objects described by point clouds. In *Canadian Conference on Electrical and Computer Engineering* (2008), pp. 1443–1448.

[18] Gopi, M., Krishnan, S., and Silva, C. Surface Reconstruction based on Lower Dimensional Localized Delaunay Triangulation. *Computer Graphics Forum 19*, 3 (Sept. 2000), 467–478.

[19] Grunberg, M., Genaud, S., and Mongenet, C. Seismic Ray-Tracing and Earth Mesh Modeling on Various Parallel Architectures. *The Journal of Supercomputing 29*, 1 (July 2004), 27–44.

[20] Gueziec, A., and Hummel, R. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics 1*, 4 (1995), 328–342.

[21] Hayward, V., and Astley, O. Performance measures for haptic interfaces. *Robotics Research* (1996), 195–207.

[22] Ho, C.-H., Basdogan, C., and Srinivasan, M. A. A ray-based haptic rendering technique for displaying shape and texture of 3D objects in virtual environments. *ASME Winter Annual Meeting 61* (1997), 77–84.

[23] Ho, C.-H., Basdogan, C., and Srinivasan, M. A. Haptic rendering: Point-and ray-based interactions. *Proceedings of the Second PHANToM Users Group Workshop* (1997).

[24] Ho, C.-H., Basdogan, C., and Srinivasan, M. A. Ray-Based Haptic Rendering: Force and Torque Interactions between a Line Probe and 3D Objects in Virtual Environments. *The International Journal of Robotics Research 19*, 7 (July 2000), 668–683.

[25] Huanlan, Z., and Guangming, Z. VC++ Implementation of Viscoelastic Ray Tracing Simulation. In *2012 International Conference on Computer Science and Electronics Engineering* (Mar. 2012), vol. 1, IEEE, pp. 37–41.

[26] Iggo, A., and Muir, A. R. The structure and function of a slowly adapting touch corpuscle in hairy skin. *The Journal of physiology 200*, 3 (Feb. 1969), 763–96.

[27] Johansson, G., and Carr, H. Accelerating marching cubes with graphics hardware. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON '06* (New York, New York, USA, Oct. 2006), ACM Press, p. 39.

[28] Josuttis, N. M. *The C++ standard library: a tutorial and reference.* Addison-Wesley, 2012.

[29] Kaluschke, M., Zimmermann, U., Danzer, M., Zachmann, G., and Weller, R. Massively-Parallel Proximity Queries for Point Clouds. In *Virtual Reality Interactions and Physical Simulations (VRIPhys)* (Bremen, Germany, 2014), Eurographics Association.

[30] Kil, Y. J., and Amenta, N. GPU-Assisted Surface Reconstruction on Locally-Uniform Samples. In *Proceedings of the 17th International Meshing Roundtable*, R. Garimella, Ed. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 369–385.

[31] Laycock, S., and Day, A. A survey of haptic rendering techniques. *Computer Graphics Forum 26*, 1 (2007), 50–65.

[32] Lee, D. T., and Schachter, B. J. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences 9*, 3 (June 1980), 219–242.

[33] Leeper, A., Chan, S., and Salisbury, K. Point clouds can be represented as implicit surfaces for constraint-based haptic rendering. In *2012 IEEE International Conference on Robotics and Automation* (May 2012), IEEE, pp. 5000–5005.

[34] Levoy, M., Ginsberg, J., Shade, J., Fulk, D., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., and Davis, J. The digital Michelangelo project. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00* (New York, New York, USA, July 2000), ACM Press, pp. 131–144.

[35] Levoy, M., Ginsberg, J., Shade, J., Fulk, D., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., and Davis, J. The digital Michelangelo project. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00* (New York, New York, USA, July 2000), ACM Press, pp. 131–144.

[36] Levoy, M., and Whitted, T. The Use of Points as a Display Primitive. 1985.

[37] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics 21*, 4 (Aug. 1987), 163–169.

[38] Matthias Teschner, Bruno Heidelberger, Matthias Mueller, Danat Pomeranets, M. G. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *VMV* (2003), pp. 47–54.

[39] McNeely, W. A., Puterbaugh, K. D., and Troy, J. J. Six degree-of-freedom

haptic rendering using voxel sampling. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99* (New York, New York, USA, July 1999), ACM Press, pp. 401–408.

[40] Nielson, G., and Hamann, B. The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceeding Visualization '91* (1991), IEEE Comput. Soc. Press, pp. 83–91,.

[41] Pascucci, V. Isosurface computation made simple: hardware acceleration, adaptive refinement and tetrahedral stripping. 293–300.

[42] Pfister, H., Zwicker, M., Van Baar, J., and Gross, M. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 335–342.

[43] Pimentel, K. 3ds Max 2013, http://area.autodesk.com/blogs/ken/3ds_max_2013_announced accessed in 2015, 2012.

[44] Ren, L., Pfister, H., and Zwicker, M. Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. *Computer Graphics Forum 21*, 3 (Sept. 2002), 461–470.

[45] Rusinkiewicz, S., and Levoy, M. QSplat. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00* (New York, New York, USA, July 2000), ACM Press, pp. 343–352.

[46] Sainz, M., and Pajarola, R. Point-based rendering techniques. *Computers & Graphics 28*, 6 (Dec. 2004), 869–879.

[47] Salisbury, K., Conti, F., and Barbagli, F. Survey - Haptic rendering: intro-

ductory concepts. *IEEE Computer Graphics and Applications 24*, 2 (Mar. 2004), 24–32.

[48] Salisbury, K., and Tarr, C. Haptic rendering of surfaces defined by implicit functions. *Proceedings of the ASME 6th Annual Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, November (1997).

[49] Samet, H. Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling).

[50] Sreeni, K. G. Distance Field based Haptic Rendering of Scattered Oriented Points. *International Journal of Computer Applications 61*, 7 (Jan. 2013), 1–8.

[51] Sreeni, K. G., and Chaudhuri, S. Haptic rendering of dense 3D point cloud data. In *2012 IEEE Haptics Symposium (HAPTICS)* (Vancouver, BC, Mar. 2012), IEEE, pp. 333–339.

[52] Srinivasan, M. A. What is haptics? *Laboratory for Human and Machine Haptics: The Touch Lab, Massachusetts Institute of Technology* (1995), 1–11.

[53] Vlasov, R., Friese, K.-I., and Wolter, F.-E. Haptic Rendering of Volume Data with Collision Determination Guarantee Using Ray Casting and Implicit Surface Representation. *2012 International Conference on Cyberworlds D* (Sept. 2012), 91–98.

[54] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June 1980), 343–349.

[55] Zhou, K., Gong, M., Huang, X., and Guo, B. Data-Parallel Octrees for Surface Reconstruction. *IEEE transactions on visualization and computer graphics* (May 2010).

[56] Zhu, W., and Lee, Y. Product prototyping and manufacturing planning with 5-DOF haptic sculpting and dexel volume updating. In *12th International Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2004. HAPTICS '04. Proceedings.* (2004), IEEE, pp. 98–105.

[57] Ziegler, G., Tevs, A., Theobalt, C., and Seidel, H. GPU point list generation through histogram pyramids. *Technical Reports of the MPI for Informatics*, June (2006), 2004–2006.

[58] Zilles, C. B., and Salisbury, K. A constraint-based god-object method for haptic display. In *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots* (1995), vol. 3, IEEE Comput. Soc. Press, pp. 146–151.